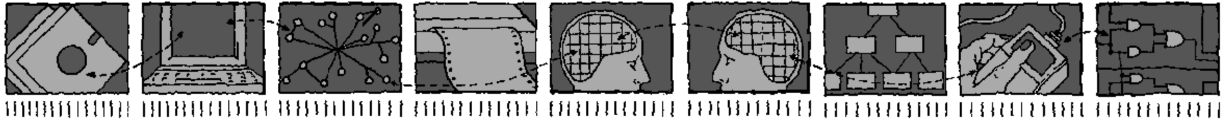
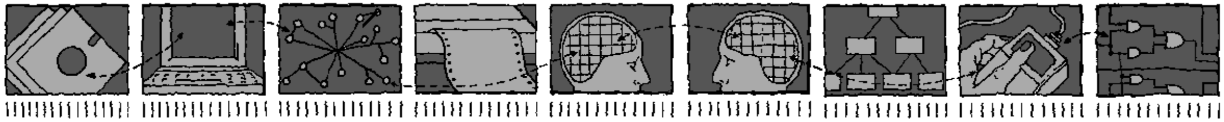


Department of Computing Science and Mathematics
University of Stirling



Formal Specification and Analysis of Digital Hardware Circuits in LOTOS

Ji He

Technical Report CSM-158

ISSN 1460-9673

August 2000

Department of Computing Science and Mathematics
University of Stirling

Formal Specification and Analysis of Digital Hardware Circuits in LOTOS

Ji He

Department of Computing Science and Mathematics
University of Stirling
Stirling FK9 4LA, Scotland

Telephone +44-786-467421, Facsimile +44-786-464551
Email jih@cs.stir.ac.uk

Technical Report CSM-158

ISSN 1460-9673

August 2000

Abstract

The thesis discusses using ISO standard formal language LOTOS (Language of Temporal Ordering Specification) for formally specifying and analysing digital circuits. The study serves two-fold: it examines the possibility of extending applications of LOTOS outside its traditional areas, and provides a new formalism to aid designing correct hardware.

Digital circuits are usually classified into synchronous (clocked) and asynchronous (un-clocked) circuits. The thesis addresses both of them. LOTOS models for signals, wires, components and component connections are established, together with the behavioural models of digital components in synchronous and asynchronous circuits. These formal models help to build the rigorous specifications of digital circuits, which are not only valuable documentation, but also the bases for further analysis. The investigation of the thesis shows that LOTOS is suitable for specifying digital circuits at various levels of abstraction. Compared with other formalisms, it is especially efficient on higher level modelling. But there is also a gap between LOTOS models and real world hardware, which is the result of the difference between inputs and outputs of systems being abstracted away in LOTOS. The gap is bridged by introducing input receptive or input quasi-receptive specifications.

Two analysis approaches are investigated in the thesis, namely formal verification and conformance testing. Verification intends to check the correctness of the formal model of a circuit, it is exhaustive and can ensure the correctness of the model being checked. While testing is applied to a physical product or a formal or informal model, it can never be exhaustive but are very useful when a formal model is difficult to build.

Current LOTOS verification techniques support the three common verification tasks, i.e. requirements capture, implementation verification and design verification. In this thesis, model checking is used to fulfill the tasks. It is found that verification of synchronous circuits is relatively straightforward since LOTOS tools can be directly used. For verifying asynchronous circuits, two conformance relations are defined to take the different roles of inputs and outputs into account. Compared with other hardware verification approaches, the approach presented in this thesis has the advantage of finding bugs at early stages of development, because LOTOS can be used in higher level modelling. Moreover, LOTOS is supported by various verification techniques, which are complementary to each others and give more chances to detect design faults.

The thesis explores a new direction of applying formal methods to digital circuit design. The basic idea is to combine formal methods with traditional validation approaches. LOTOS conformance testing theory is employed to generate test cases from higher level formal specifications. The test cases are then applied to commercial VHDL (VHSIC Hardware Description Language) simulators to simulate lower level circuit designs. Case studies reveals that the approach is very promising. For example, it can detect bugs which cannot be captured by examining a formal model.

Timing characteristics are important factors in digital design. To be able to specify and analyse timed circuits, ET-LOTOS is exploited. Two important timing characteristics in digital circuits, namely delays and timing constraints are identified. Timed specifications of digital circuits are the composition of these timing characteristics and functionality. Based on the formal specifications, rigorous analysis can be applied. The method is valuable in discovering subtle design bugs related to timing, such as hazard, race conditions, and can also be used for analysing speed performance of digital circuits.

Contents

1 Introduction **1**
1.1 Motivation . . . 1
1.2 Context of the Research . . . 1
1.2.1 Design Procedure for Digital Circuits . . . 1
1.2.2 Formal Methods in Digital Circuit Design . . . 3
1.3 Advantages of Using LOTOS in Digital Circuit Design . . . 4
1.4 Thesis Structure . . . 5

2 LOTOS and Overview of DILL **7**
2.1 A Brief Introduction to LOTOS . . . 7
2.1.1 Basic LOTOS . . . 7
2.1.2 Data Types . . . 9
2.1.3 Full LOTOS . . . 10
2.1.4 Semantics of LOTOS . . . 11
2.2 Overview of DILL . . . 12
2.2.1 General Consideration . . . 12
2.2.2 Underlying Modelling Approach . . . 12
2.3 Conclusion . . . 15

3 Specification of Digital Logic Circuits **17**
3.1 Background . . . 17
3.1.1 Synchronous Circuits and Asynchronous Circuits . . . 17
3.1.2 Delay and Environmental Models . . . 18
3.2 The First Model of Basic Logic Gates . . . 19
3.3 Specifying Synchronous Circuits . . . 20
3.3.1 Structure of Synchronous Circuits . . . 20
3.3.2 Basic Logic Gates in Synchronous Circuits . . . 21
3.3.3 Specifying Storage Elements . . . 22
3.3.4 Specifying Circuit Behaviour . . . 23
3.3.5 Case Study: Specifying a Single Pulser . . . 24
3.4 Specifying Asynchronous Circuits . . . 25
3.4.1 Classes of Asynchronous Circuits . . . 25
3.4.2 DILL and Speed Independent Circuits . . . 26
3.4.3 Basic Logic Gates in SI Circuits . . . 26
3.4.4 Other Basic Building Blocks of SI Circuits . . . 28
3.4.5 Input Receptiveness . . . 29
3.4.6 Input Quasi-Receptiveness . . . 30
3.4.7 Case Study: Specifying a FIFO . . . 32
3.5 Related Work . . . 34
3.6 Conclusion . . . 35

4 Verification of Digital Logic Circuits **36**
4.1 Background . . . 36
4.1.1 What is to be verified . . . 36
4.1.2 Temporal Logic and ACTL . . . 37
4.1.3 Relations between LTSs . . . 39
4.1.4 CADP . . . 41
4.2 Verification of Synchronous Circuits . . . 42
4.2.1 Verifying Properties . . . 42
4.2.2 Verifying Relations between Two Levels . . . 43
4.2.3 Case Study . . . 43
4.3 Verification of Asynchronous Circuits . . . 47

4.3.1 Extra Considerations for Verifying Asynchronous Circuits . . . 47
4.3.2 Environment of Asynchronous Circuits . . . 48
4.3.3 Conformance and Strong Conformance . . . 50
4.3.4 Case Studies . . . 52
4.3.5 Asynchronous FIFO . . . 52
4.4 Related Work . . . 54
4.5 Conclusion . . . 56

5 Testing Digital Logic Designs **57**
5.1 Background . . . 57
5.1.1 Testing and Verification . . . 57
5.1.2 Formal Conformance Testing for LTSs . . . 57
5.1.3 Overview of the Approach . . . 58
5.1.4 Conformance Testing for IOLTS . . . 59
5.1.5 IOCONF and IOCO . . . 59
5.1.6 Test Generation . . . 62
5.2 Application to Synchronous Circuits . . . 63
5.2.1 DILL Specifications of the Examples . . . 63
5.2.2 LTSs, Suspension Automata and Test Cases . . . 64
5.2.3 Test Generation and Execution . . . 65
5.2.4 Further Discussion . . . 68
5.3 Application to Asynchronous Circuits . . . 68
5.4 Case Study . . . 71
5.5 Related Work . . . 72
5.6 Conclusion . . . 75

6 Specification and Analysis of Timed Circuits **76**
6.1 Background . . . 76
6.1.1 ET-LOTOS in Brief . . . 76
6.2 LOTOS Model of Timed Circuits . . . 77
6.3 Specifying Functionality . . . 78
6.4 Modelling Delays . . . 79
6.4.1 Basic Delay Types . . . 79
6.4.2 Inertial Delay . . . 79
6.4.3 Pure Delay . . . 80
6.4.4 General Delay . . . 81
6.4.5 Delay Components for Higher Level Specifications . . . 82
6.5 Modelling Timing Constraints . . . 83
6.6 Case Study: A 2-to-1 Multiplexer . . . 84
6.6.1 TE-LOLA . . . 84
6.6.2 Behavioural Specification and Validation . . . 85
6.6.3 Structural Specification and Validation . . . 86
6.7 Related Work . . . 87
6.8 Conclusion . . . 88

7 Conclusion **90**
7.1 Main Contributions . . . 90
7.2 Future Work . . . 91

A Glossary **101**

B DILL Library Components **103**

Acknowledgment

This thesis would not have been written without the help of many people.

I would like to express my gratitude to professor Kenneth J. Turner, my supervisor, for creating the opportunity of my studying in the U.K. I thank Ken for the substantial time and expertise he has contributed to my thesis, and for his consistent encouragement during the past three years. I am grateful for his patience in correcting all those ungrammatical sentences in the drafts of my reports, our joint papers and this thesis. I lack the room here to detail all the things he has done on my behalf.

I would like to thank professor Leslie Smith for agreeing to be my second supervisor, for reading the drafts of some of my reports, and for his attention on the progress of my research work. I also thank him for his contributions and time as my Ph.D examiner.

The department of Computing Science and Mathematics in Stirling is an excellent place to work in. I thank all the members there for their friendliness to me. I would like to thank Susie Whitelam, Moira Taylor, Antonia Servera and Heather Brennan, who make my life in Stirling easier and unforgettable. Thanks are due to Sam Nelson and Graham Cochrane, who are always ready to provide technical support. I would like to thank Frank Kelly for providing me with the circuits he designed as my case studies, and drawing circuit diagrams for my first technical report. I am grateful for the encouragement from Savi Maharaj and Carron Shankland, they also carefully read the drafts of my papers and gave me valuable comments on the contents. I also thank Carron for allowing me to use her new Sun workstation, which might be the most powerful machine in the department currently.

I would like to thank professor Muffy Calder of Glasgow University for several fruitful discussions with me, and her contributions and time as my external examiner.

I would like to thank Jan Tretman of University Twente, Netherlands, for providing me with his papers related to his theory of testing Input Output Transition Systems, and explaining some details of the theory to me.

I would like to thank the developers of the tool CADP in Inria, France. In particular Hubert Garavel, Laurent Mounier, Mihaela Sighireanu and Radu Mateescu, who are always quick in replying my questions.

Thanks are also due to professor Tom Melham of Glasgow University, and professor Graham Birtwistle of Leeds University, who kindly gave suggestions on my research work at the early stage of my Ph.D study.

I would especially like to thank my mum for her love and long lasting support which I might never be able to reward. I would like to thank my sisters and their families for their friendship and help. Special thanks to my husband, Zhongming, for his understanding and support, and for his criticism on the drafts of this thesis, which have no doubt helped to improve it greatly.

This work was supported by ORS (Overseas Research Studentship), Sino-British Fellopship Trust, and University of Stirling.

List of Figures

1	Design procedure for digital circuits	2
2	A LOTOS behaviour expression and its behaviour tree	8
3	Connecting *And* gates with two *Inverters*	14
4	Structure of Reset Set Latch and its truth table	15
5	Inertial and pure delay	19
6	Structure of synchronous circuits	20
7	An implementation of a *Nand2* gate	21
8	Examples of cyclic connections	22
9	Single pulser design	25
10	An isochronic fork	25
11	Modelling DI, QDI as SI	27
12	Two wires in series	30
13	LTS with internal events	31
14	FIFO with one stage and with two stages	33
15	Implementation of a cell of FIFO	33
16	Different equivalence relations with different strengths	41
17	The LTSs of single pulser specifications	43
18	A bus arbiter with three cells	44
19	Design of an arbiter cell	45
20	LTSs of the Specification and implementation of *Repeater*	48
21	Specification vs Environment	49
22	Input quasi-receptive environment	49
23	Non-deterministic specification and its implementations	50
24	The suspension automata of LTSs in figure 23	52
25	A circuit with two elements	54
26	Quasi-receptive specifications of components in figure 25	55
27	DILL approach of testing circuit designs	60
28	The difference between IOCONF and IOCO	61
29	The LTS of the JK flip flop and Single-Pulser	64
30	Suspension automata of JK flip flop and Single-Pulser	64
31	Several tests of JK flip flop and Single-Pulser	65
32	LTS, suspension automaton and several tests of FIFO	69
33	LTS, suspension automaton and one test of Selector	70
34	Nodes with more than one outputs and its test trace	70
35	The controller of the Black-Jack Dealer	72
36	The datapath of the Black-Jack Dealer	73
37	Control-Datapath communication	74
38	The specification model for a timed component	78
39	Basic delay types	79
40	Catch-Up phenomenon with pure delay	81
41	D Flip-Flop with asynchronous Pre-Clear	82
42	Setup and hold times for D Flip-Flop	83
43	Width and Period timing constraints	84
44	Structure of 2-to-1 multiplexer as timed logic gates	86
45	Hazards and their LOTOS specifications	86
46	The hazard-free Multiplexer	87

List of Tables

1 A counter-example generated by Aldébaran . . . 46
2 Test suite for JK flip flop . . . 67
3 Two test cases for Single Pulser . . . 68
4 Test suite of FIFO . . . 71
5 Test suite of Selector . . . 71
6 Hazards in the 2-to-1 Multiplexer . . . 87
7 The components of synchronous circuits in the DILL library . . . 103
8 The components of asynchronous circuits . . . 103
9 Timed components in the DILL library . . . 104
10 Selected LOTOS syntax . . . 105
11 Seclected ET-LOTOS syntax . . . 106

1 Introduction

1.1 Motivation

This thesis is concerned with using ISO standard formal language LOTOS (Language Of Temporal Ordering Specification, [ISO89]) to formally specify and analyse digital circuits. It has a two-fold purpose: examining the possibility of applying LOTOS outside its traditional area, and providing new theories and tools to aid designing correct hardware.

LOTOS has been widely and successfully used to specify communication systems such as standards for OSI (Open Systems Interconnection [ISO94]). This is not surprising since LOTOS was developed for this purpose. It has also been used in related area such as Open Distributed Processing [ISO95]. However, LOTOS might claim to be a general-purpose language for specifying concurrent systems, so it is valuable to investigate the applicability of the language outside its original field. Digital circuits are complex systems which involve intensive concurrency. The application of LOTOS in this new area will help to discover the strengths and limitations of the language.

Although digital logic design is well understood, guaranteeing the correctness of a circuit is still a very hard problem. Formal methods provide a solution by systematically and exhaustively analysing circuit behaviour to prove the correctness or pinpoint the bugs. Many formalisms have been used to model digital circuits, including HOL (Higher Order Logic [MGG93, SRI91], process algebra [MM92], automaton [HHK96, BCM+92], functions [O'D95] and Petri Nets [Rei85, YK98]. As an internationally standardised formal language, LOTOS should be more easily accepted by industry. It is more expressive than most formalisms developed for academic research, and is supported with theory and tools that allow various analysis methods, some of which are not possible with other hardware specification approaches. The use of LOTOS is alternative and complementary to the existing methods for designing correct hardware.

Following the initial investigation of the subject in [TS94], the approach presented in this thesis is named DILL (DIgital Logic in LOTOS).

1.2 Context of the Research

1.2.1 Design Procedure for Digital Circuits

Design of digital circuits is a complicated process which involves many different steps. Figure 1 depicts a typical design flow [GDKW92] used in industry.

Design starts with an initial idea, which is abstract and may be recorded in diagrams or a natural language. Human designers have to build a specification of the idea in some higher-level description language, such as Verilog [IEE95], VHDL [IEE93], LOTOS, or other formalism such as finite state machines etc.

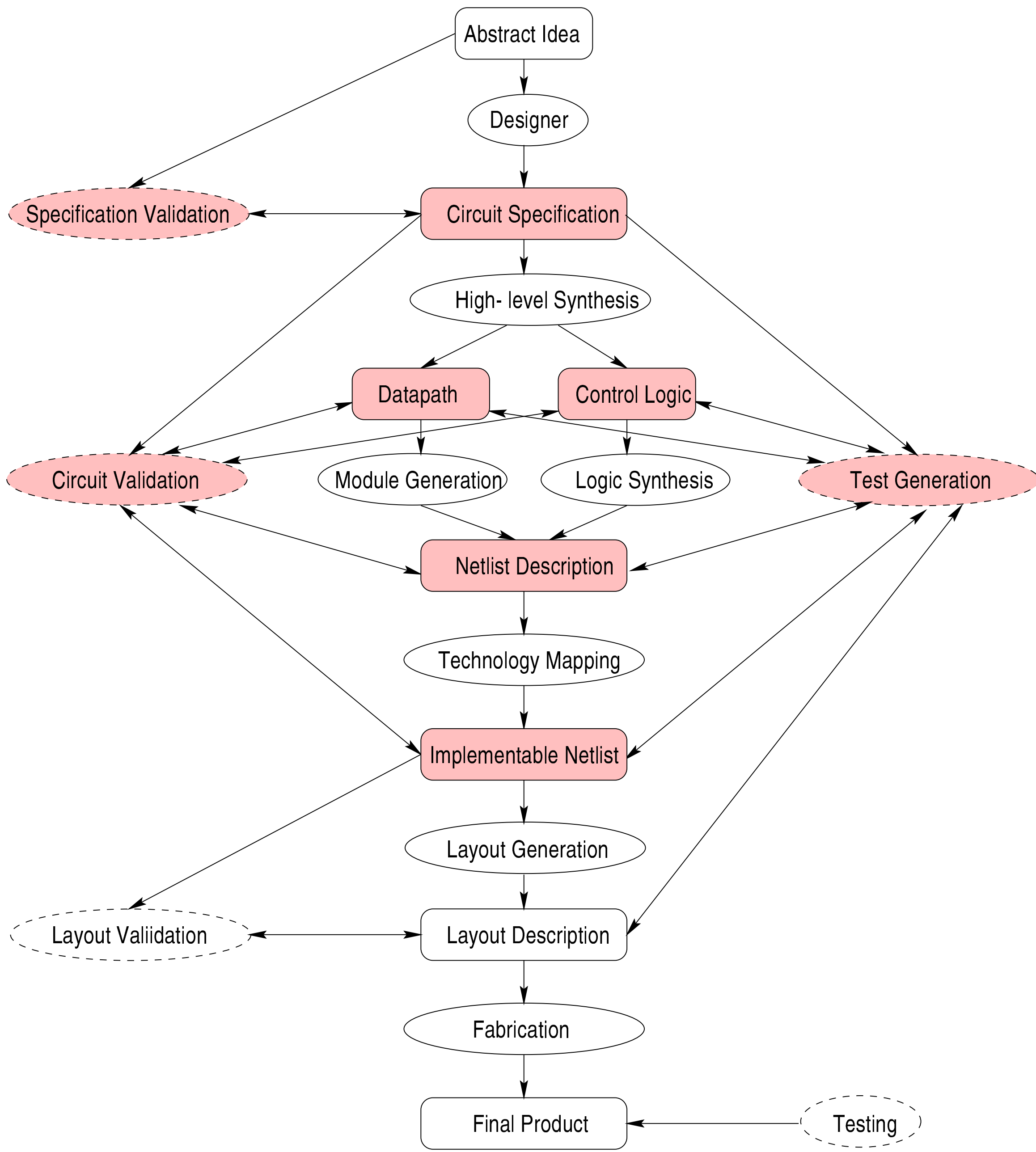
This higher level circuit specification is then refined to a *register transfer level* (RTL) specification by human designers or high level synthesis tools. Typically an RTL specification contains two parts: datapath and control logic. The datapath is built from elements such as registers, multiplexers, adders, multipliers, etc. The control logic provides necessary control signals and timing for the datapath.

Logic synthesis tools are then applied to generate a description of the control logic in the form of a netlist of gates. These gates correspond to a set of logic equations and may not be physically implementable.

Given these gates, technology mapping replaces them with the implementable gates in a certain library. It also combines small gates to make larger ones as long as they are available in the library. Components in datapaths are usually mapped into the descriptions at transistor level directly, using the tool called a module generator.

The lowest level description is the layout of the circuit. Modules and gates are placed and connected by using placement and routing tools. The layout can be sent for fabricating to get the final product.

The design procedure inevitably involves iterations, either because performance requirements are not met or errors appear after a transformation. Human mistakes contribute most of the errors, but other sources are also possible, such as deficiencies in synthesis algorithms or software bugs. Although many steps in the design flow can be done automatically, human design is unavoidable. For example, high level



Rounded rectangles represent the different descriptions of a circuit.
Ellipses with a solid edge identify the processes of transforming one description to the other
Ellipses with a dashed edge stand for validation methods used to ensure the correctness of each design step.
Shaded parts are in the scope of the investigation of this thesis.

Figure 1: Design procedure for digital circuits

synthesis is still an area of current research and not widely used in industry. Logic synthesis is a well-established technique but is not suitable for regular blocks such as RAM (Random-Access Memory), PLA (Programmable Logic Array) and complete microprocessors because of the speed and area inefficiency.

Validation tools are therefore necessary for detecting design errors. Checking the correctness of a higher level circuit specification against an initial design idea is termed *specification validation*, and examining the correctness of the design at RTL and gate level (including the netlist description and the implementation netlist) is termed *circuit validation*. *Layout validation* tools are dedicated to the circuit specified at layout level. While *testing* usually refers to the activity of detecting fabrication deficiencies in final products.

At present, simulation is still the predominant approach of validating digital designs in industry. Test vectors are fed into an executable description of the circuit; the behaviour of the circuit is compared with that of a reference model, or simply analysed by observing the outputs. The procedure for obtaining test vectors is called *test generation*. Most test vectors for simulation are created by experienced testing engineers. Test vectors for testing fabrication bugs are usually generated automatically, with the assumption that only certain kinds of faults may occur in the process of manufacturing.

The thesis supports earlier stages of the design procedure. More precisely it mainly concerns the correctness of logic designs rather than the correctness of layout or fabrication. Automatic synthesis of circuits is excluded from the scope of the thesis either, but some related work of hardware synthesis from LOTOS specification can be found in [HYKT94]. The shaded parts in figure 1 are therefore in the scope of the thesis. In other words the thesis investigates how to specify circuits at behavioural, RTL, and gate levels. With respect to validation, it covers specification validation, circuit validation and test generation.

1.2.2 Formal Methods in Digital Circuit Design

Today's digital circuits are so large and complex that testing engineers can no longer create all the vectors required in order to examine circuits adequately with simulation approach. Moreover simulation run time has increased even faster than circuit size. For larger chips, it is measured in days or even weeks. The iterations in the design process require that every change should be checked thoroughly, but time pressure forces only a subset of test vectors to be used for each revision. This exposes the design to a great risk of errors.

The weakness of simulation spurs research in applying formal methods in digital designs. Formal methods refers to `*mathematically-based languages, techniques and tools for specifying and verifying systems*'[CW96]. In other words, it contains two aspects: *formal specification* and *formal verification* (or *verification* for short). Formal specification uses a language or notation with a mathematically-defined syntax and semantics to describe a system and its desired properties. Formal verification is the approach to prove the correctness of the specification. The advantages of verification over simulation are that it is exhaustive in the sense that all the behaviour of the model of a system will be checked, and that it is faster than simulation in many cases since the result might be obtained after a single run of the verification program.

Although formal verification has been the main theme of using formal methods in hardware design, it is not possible without proper specifications. But formal specification is not just the base for verification; writing things down precisely itself is valuable: a deeper understanding of the specified circuit can be obtained, and the inconsistencies, ambiguities or flaws in the initial idea can be discovered. These incompletenesses will become very difficult and expensive to detect when they are transformed to lower level designs. Specification also serves as a permanent documentation of the requirements, the behaviour and the implementation of a circuit. It is a precise and convenient bridge between the various parties involving in the design, implementation and use of the circuit. So far many formalisms have been used to specify digital hardware, including HOL, process algebra, automata, functions, and Petri Nets. Apart from specifying circuits in a formal language or notation, an alternative approach is to provide formal semantics for an ordinary hardware description language (HDL), but usually only a subset of an HDL is suitable for formalization. The most popular industry HDLs, namely VHDL, Verilog and ELLA [MC93], have been studied using this method [KB95, Gon95, BGMW95].

There are essentially two approaches to formal verification of digital circuits: model checking and theorem proving. Model checking is a technique that is based on constructing a finite model of a system and checking that whether a desired property holds on the model. The check is performed by an exhaustive

state space search. Since the model is required to be finite, model checking is guaranteed to terminate. Two styles exist for model checking. In *temporal logic model checking*, properties are specified in a temporal logic [Pnu77] and a circuit is modelled as a finite state transition system. Efficient search procedures have been developed to see if temporal logic formulae hold on the model. In *conformance checking*, both desired properties and the implementation of a circuit are modelled as automata or labelled transition systems (LTS). Then the two models are compared to determine whether or not the implementation conforms to the properties. Various notions of conformance have been investigated, such as language inclusion [FK97] and observational equivalence [Mil89]. The advantages of model checking lie in that it is completely automatic and fast, and that it produces counterexamples when verification fails, which is particularly useful in practice. The disadvantage of model checking is the state space explosion problem. Many approaches have been proposed to tackle this problem, such as BDD (Binary Decision Diagrams [Bry92]) and localization reduction [FK97].

Verification by theorem proving relies on expressing both the system and properties as formulae in some mathematical logic. This logic is given by a set of axioms and inference rules. Theorem proving is the process of finding a proof of a property from the axioms and inference rules. The benefits of this approach are that it is general and can deal with systems with infinite state space. But generating a proof automatically is very difficult in theory and in practice. In addition, theorem proving is able to *prove* correctness but is unable to pinpoint the errors in incorrect designs. These two limitations prevent this method from being used widely in industry.

Although substantial progresses in formal verification has been achieved over the last decade, the size of the circuit which can be successfully analysed by verification is still considerably smaller than the size of the circuit which can be manufactured. Simulation is still the most broadly adopted approach in industry. One of the new research areas is to combine the traditional simulation-based validation with formal methods, as suggested in [Dil98]. An approach proposed in this thesis is in line with the idea. Unlike in the current design flow, where test vectors are written by experienced test engineers, they are generated automatically from a formal specification of the behaviour of the circuit. These test vectors are then used as stimuli in a conventional simulation environment. Compared to the traditional method, the approach saves human resource and time in designing test cases, and guarantees reasonable coverage of the generated test vectors. Compared to formal verification, it avoids building the state space of a lower level specification of the circuit, which is much larger than the state space of a higher level specification. This makes it possible to find bugs which cannot be found by checking some formal models. It is hoped that the results of thesis will encourage more research in a similar area.

1.3 Advantages of Using LOTOS in Digital Circuit Design

Compared to traditional HDLs, the formal basis of LOTOS supports rigorous specification and analysis in a way that semi-formal languages (e.g. VHDL) do not. The semantics of current HDLs used in industry is based on simulation, which offers little help for thorough analysis of a circuit behaviour. Although formal semantics has been defined for some of these languages, it usually covers only a small subset of a language, and the subset is much less expressive than the original one.

LOTOS can be used in a wide-spectrum manner at a number of levels of abstraction. This allows a consistent formalism to be used during hardware design, from the high-level architecture down to the component or gate level. Refinements between levels can be checked using standard LOTOS verification techniques.

Designed for industry usage, LOTOS is more expressive than most formalisms created for research, such as CSP (Communicating Sequential Processes [Hoa85]), CCS (Calculus of Communicating Systems [Mil89]), CIRCAL (Circuit Calculus [MM92, Mil95]). In fact, the research reported here is inspired by the success of CIRCAL, a process algebra designed for specifying and analysing digital circuits. Compared to CIRCAL, LOTOS specifies digital circuits not only at relatively lower levels (e.g gate level, RTL level) but also at higher levels, such as algorithmic or system requirement level. LOTOS is therefore more suitable for specifying real-world circuits.

The formal basis of LOTOS allows verification of hardware designs. LOTOS inherits a well-developed theory of equivalences and relations from the field of process algebra and has a well-developed theory of testing and test derivation (e.g. [Bri87]). This offers interesting alternatives to other validation approaches.

Being an international standard, LOTOS is well supported by general-purpose toolsets such as CADP (Cæsar/Aldébaran Development Package [FGM+92]), LOLA (LOTOS Laboratory [QPF89]) and LITE (LotoSphere Integrated Tool Environment [van91]). All these tools can be directly used for hardware verification or simulation, therefore efforts on tool development can be substantially reduced.

LOTOS is neutral with respect to whether a specification is to be realized in hardware or software. At a high level of abstraction, the same specification may ultimately be implemented in either way. This allows LOTOS to be used for hardware-software co-design [SLM+96]. LOTOS is thus more general than a pure hardware description language.

1.4 Thesis Structure

Chapter 2 comprises two parts. It first briefly introduces the specification language LOTOS. Then gives an overview of DILL. This includes some general considerations for DILL system, and the basic modelling techniques used, e.g. how to represent signals, wires, components, and how to write specifications of the behaviour and structure of a circuit. This chapter serves as a starting point for the remaining chapters in the thesis.

Chapter 3 presents the specification of digital circuits in DILL. Synchronous circuits and asynchronous circuits are both considered. For specifying synchronous circuits, their typical structure is described first, then the models of the components in the structure are presented. The chapter also illustrates how to specify circuits at different levels of abstraction. For specifying asynchronous circuits, different types of asynchronous circuits are introduced. In this chapter only those which assume unbounded delays are considered. Besides basic logics gates, other common components used in asynchronous circuits are also specified. Finally the chapter introduces the concept of *input receptiveness* and *input quasi-receptiveness*, which are important for faithfully modelling the behaviour of asynchronous circuits.

Chapter 4 goes one step beyond specification. It presents the DILL approach to verifying digital circuits. Three common verification tasks, i.e. *requirements capture, implementation verification* and *design verification* are introduced. In DILL, equivalence and preorder checking of two LTSs are employed for implementation verification, and ACTL (Action based Computation Tree Logic [DV90]) temporal logic model checking is used for the other two tasks. A synchronous benchmark circuit, the *Bus Arbiter*, is specified and verified to illustrate the approach. The chapter also discusses the differences between verifying asynchronous and synchronous circuits. Two novel relations between LTSs are then defined for implementation verification of asynchronous circuits. As will be discussed, these relations provide intuitive criteria of correctness of asynchronous circuits. A verifier *VeriConf* is also implemented for checking the relations. Part of this chapter has been published in [JT99b].

Chapter 5 explores a new direction in applying formal methods to digital circuit design. The foundation of the chapter is the theory of testing input-output transition systems (IOLTSs) [Tre96], an extension of traditional LOTOS testing theory. Following the introduction of the theory, the chapter illustrates the suitability of applying it to generating test cases for digital circuits. To achieve satisfactory coverage of the test cases, an algorithm based on a transition tour of the state space graph is developed and implemented in a test generation tool *TestGen*. A testbench is also developed to supply these tests to a conventional VHDL simulator automatically. Finally a benchmark circuit, the *BlackJack Dealer*, is studied to examine the approach. Part of this chapter has been published in [JT99a].

Chapter 6 uses ET-LOTOS (Enhanced Timed-LOTOS [LL94]) to write circuit specifications which contain quantitative timing magnitudes. Timing information is abstracted away in the previous chapters, but it is a critical factor in deciding the correctness of circuits as well as their performance. This chapter first identifies the important timing characteristics in digital circuits, namely *timing constraints* and *delays*. Various timing constraints and delays are then specified in ET-LOTOS, including setup time, hold time, period of clock, pure delay, inertial delay, etc. The chapter also develops a model for timed specification of circuits, which has a nice property that the untimed components in previous chapters are a special case of the timed ones. This chapter uses the tool TE-LOLA (Time Extended LOTOS Laboratory [PLR95])to specify and analyse timed specifications of circuits.

Chapter 7 summarizes the thesis and presents some overall conclusions.

Appendix A contains the glossary.

Appendix B summarize the components in DILL library.

Appendix C contains the syntax of LOTOS.
Appendix D contains the syntax of ET-LOTOS.

2 LOTOS and Overview of DILL

This chapter briefly introduces the formal specification language LOTOS and gives an overview of the DILL approach. General considerations in the DILL approach are explained, followed by the discussion of the basic modelling approach adopted. This includes how to model signals, wires and digital components, as well as how to specify the behaviour and structure of circuits.

2.1 A Brief Introduction to LOTOS

LOTOS is a formal language standardised by ISO in 1989 (ISO 8807) for the design of OSI services and protocols. The name reflects the fact that LOTOS describes a system by defining the order in which the events of the system may occur. LOTOS is made up of two parts. The first part is used to specify system behaviour and is derived from process algebra, mainly from CCS and CSP. The second part defines abstract data types and is based on the language ACT ONE [EM85]. The process algebra aspect of LOTOS is called *basic* LOTOS, the combination of basic LOTOS with data types is termed *full* LOTOS. The following sections present the aspects of the language which are required in the thesis.

2.1.1 Basic LOTOS

In LOTOS a system and its components are represented as *processes*. A process interacts with its environment through *gates*, and displays its behaviour in terms of permitted sequences of actions. These actions, termed *events* in the LOTOS terminology, are the results of the interactions of a process and its environment. Each event is associated with a gate, namely the gate at which the event happens.

The behaviour of a system is described in LOTOS by a *behaviour expression*, a language construct in which the sequences of allowed events are defined. Behaviour expressions can be illustrated as *behaviour trees*. In these trees, a node represents a state of a system. An arc between nodes represents a transition which causes the system to move from one node to another, and is labelled with the corresponding event. For clarity, arrows may be added to arcs to indicate the beginning and the end states of transitions. See figure 2 for an example of a behaviour tree, which is a two-key system designed by Quemanda [Tur93]. LOTOS provides the following basic operators to build language constructs:

- Inaction (**stop**)

  Inaction models a situation where a process is unable to interact with its environment. It is also called *deadlock*. In behaviour trees, inaction corresponds to a node that does not lead to any arcs.

- Action Prefix (**;**)

  Action prefix is used when an event must occur before other behaviour expressions. If *a* is an event and *B* is a behaviour expression, *a; B* denotes that *a* must happen before behaviour *B*. In behaviour trees, action prefix is illustrated with two nodes and an arc which is labelled with the action.

- Choice (**[]**)

  The choice operator denotes that two alternative and exclusive behaviours can happen. If *B1, B2* are behaviour expressions, *B1* **[]** *B2* behaves as *B1* or *B2* depending on whether the next event provided by the environment is the initial one of *B1* or *B2*. If the two have the same initial event, the system behaves non-deterministically. Choice is represented by branches in behaviour trees.

- Internal Events (**i**)

  Internal event **i** is a special LOTOS event which represents the actions that are internal to a system and therefore invisible to its environment. Internal events may also introduce non-determinism.

- Termination (**exit**)

  Exit models the successful termination of processes. The interpretation of **exit** is that a special success event (called δ) takes place before **stop**.

```
hide In2, Out2 in
In1; in2; Access;
(Out1; Out2; stop  [] Out2; Out1;stop  )
[]
In2; In1; Access;
(Out1; Out2; stop  []  Out2; Out1;stop  )
```

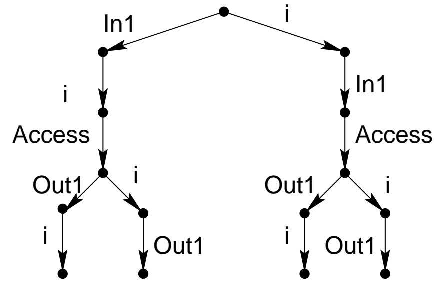


Figure 2: A LOTOS behaviour expression and its behaviour tree

- Parallel Composition (|**[]**|)

  If *B1, B2* are behaviour expressions and *g1, g2, ⋯, gn* represent gates, then *B1* | $[\mathbf{g1}, \mathbf{g2}, \cdots, \mathbf{gn}]$ | *B2* represents that events at the gates that belong to $g1, g2, \cdots, gn$ can occur only with the participation of both *B1* and *B2*. Other events take place with the participation of *B1* or *B2* alone. In other words, $g1, g2, \cdots, gn$ are the gates at which *B1, B2* synchronize. For example *a; b;* **stop** | [**b**] | *c; b;* **stop** can either behave as *a; c; b;* **stop** or *c; a; b;* **stop**.

  There are two special cases for parallel composition, namely *pure interleaving* (|||) and *full synchronization* (||). ||| is the shorthand for | [] |, i.e. no synchronisation is required, each system behaves at its own pace. While || is the shorthand for | $[\mathbf{g1}, \mathbf{g2}, \cdots, \mathbf{gn}]$ |, where $g1, g2, \cdots, gn$ are all the gates appearing in *B1* and *B2*; this means that *B1* and *B2* have to synchronize at all the gates.

  LOTOS supports *multi-way synchronisation*, meaning that more than two processes can synchronize at a gate. If *P [a, b, c], Q [a, c], R [a, b]* are three processes representing three components of a system, then *P [a, b, c]* | [**a**] | *Q [a, c]* | [**a**] | *R [a, b]* says that events at gate *a* can happen only with the participation of processes *P, Q* and *R*.

- Hiding (**hide**)

  otosThe *hide* operator provides the mechanism of abstraction. If *B* is a behaviour expression, **hide** *g1, g2, ⋯, gn* **in** *B* makes gates *g1, g2, ⋯, gn* invisible to the environment. Interactions on these gates therefore become internal events.

- Sequential Composition (>>)

  Sequential composition represents temporal ordering of behaviour. If *B1, B2* are behaviour expressions, then *B1* >> *B2* expresses that *B2* occurs after *B1*, provided that the special event δ has appeared in *B1*. Recall that action prefix is used to represent the temporal ordering of events (not behaviour expressions).

- Disabling ([>)

  Disabling represents that the behaviour of a system is disrupted by an exceptional circumstance. If *B1, B2* are behaviour expressions, then *B1* [> *B2* behaves like *B1* until the initial event of *B2* happens, then it behaves like *B2*. If *B1* terminates successfully, *B2* does not apply.

Figure 2 gives an example of a behaviour expression and its behaviour tree, quoted from [Tur93].

A *process declaration* can then be written based on these language constructs to represent the behaviour of a system. It is delimited by the reserved words **process** and **endproc**. A process is made up of a name, a possible formal gate list, a possible formal parameter list, and a behaviour expression. In the behaviour expression of a process, it is allowed to declare nested processes to introduce sub-systems, which are preceded by the reserved word **where**. Reserved words **exit** or **noexit** are used to indicate if the process can terminate successfully or not. The following is an example of process declaration.

```
process P [a, b, c] : noexit :=                                   1
  P1 [a, b]                                                        2
```

```
|[a]|                                   3
 P2 [a, c]                              4
                                        5
where                                   6
 process P1 [aa, bb] : noexit :=        7
  aa; bb; P1 [aa, bb]                   8
 enproc                                 9
                                        10
 process P2 [aa, cc] : noexit :=        11
  aa; cc; P2 [aa, cc]                   12
 endporc                                13
endproc                                 14
```

When a process is instantiated, actual gates and parameters should be provided. For example *P1 [a, b]* in line 2 is the instantiation of the declaration *P1 [aa, bb]* in line 7. Note that *P1* and *P2* refer to themselves respectively, this is a way to express recursive behaviour in LOTOS. This example does not illustrate the parameter lists since LOTOS data type are required to introduce formal and actual parameters.

A **specification** is a special process which represent the whole system, more details about specifications will be presented in section 2.1.3.

2.1.2 Data Types

LOTOS models data as *abstract data types* (ADT). The word *abstract* refers to the fact that properties of a data type are defined by the specifier rather than predefined in the language. In other words no particular implementation of data types are implied by the language.

A data type is defined by three parts: *sorts* define the sets of values of the data type, *operations* declare the operators to manipulate the data values, and *equations* define the semantics of the operations by stating which expressions are considered equal. In LOTOS, a type may be extended to define a new one by adding new sorts, operations or equations, and several types may also be combined to form a more complex one.

There are effectively no predefined data types in LOTOS. But commonly required data types can be included from the *standard library* defined in ISO 8807. These standard data types include **Boolean**, **Bit**, **NaturalNumber**, **Set** and **String** etc. The following is a small example which defines two constants and two logic operations. It is an extension of the data type *Bit* defined in the standard library. Note that LOTOS allows overloading of operators. *(* *)* introduces comments in LOTOS.

```
library
  Bit                          (* including the type Bit in library *)
endlib
type BitOp is Bit                      (* define a new type BitOp *)
  sorts BitOp                     (* the new sort is called BitOp *)
  opns                                     (* declare operations *)
    and          : -> BitOp        (* and is a constant of sort BitOp *)
    or           : -> BitOp         (* or is a constant of sort BitOp *)
    _and_, _or_ : Bit -> Bit    (* and, or are binary, infix operators *)
  eqns                                     (* define the equations *)
    forall b : Bit                     (* variable b has the sort Bit *)
    ofsort Bit          (* the range of the equations has the sort Bit *)
    b and 0 = 0;
    b and 1 = b;
    b or 0 = b;
    b or 1 = 1;
endtype                             (* end of the type definition *)
```

2.1.3 Full LOTOS

The combination of basic LOTOS with data types makes LOTOS more expressive. The following lists some of the language constructs which are offered by full LOTOS.

Action prefix may be associated with *experiment offers* and *selection predicates*. There are two kinds of experiment offers: *value offers* are in the form *! v* where *v* is a value expression, while variable offers are in the form *? x: s* where *x* is a variable of a sort *s*. A selection predicate may follow experiment offers to impose conditions on the value being offered. For example *G ? x : Bit [x = 0];* is an action prefix in which *? x : Bit* is the experiment offer and *[x = 0]* is the selection predicate, meaning than an event occurs at gate *G* and it offers a variable of sort *Bit* which should be equal to *0*.

An event can take place only when the experiment offers provided by each participating process can match each other. The most common matching used in DILL is *value passing*, where a value offer matches a variable offer. Consequently the variable receives the value supplied by the value offer. For example *G ? x : Bit* can match *G ! 0* and *x* receives value *0*.

Apart from the selection predicates mentioned above, *Guards* also impose conditions on the behaviour of a system. For example *[x = y] − > B* indicates that behaviour *B* occurs only if *x* is equal to *y*.

Parameterised **exit** and sequential composition allow values to be conveyed from one successfully terminating behaviour expression to the subsequent one. In the following behaviour expression, the input *x* is either delivered to the output or not depending on the value of *x*. The parameter of **exit** may also be **any**, in which case any value of the sort is allowed to be conveyed.

```
 (input ? x : Bit; exit (x))
>>
 (accept y : Bit in
  [y = 0] -> output ! y; exit
 []
  [y = 1] -> exit
 )
```

A *local value definition* associates values with free variables in a behaviour expression. It resembles the assignment statement in ordinary programming languages. For example, **let** *x : Bit = newIp* **in** *B* associates variable *x* with the expression *newIp* in behaviour expression *B*.

A specification may comprise two parts. In the optional *global type definition* part, data type definitions which are accessible to the overall behaviour expression, other data type definitions and processes are specified. The reserved word **behaviour** introduces the behaviour part of a specification, which is made up of a behaviour expression and some other possible process declarations, the later being referred in the former. The difference between a specification and a process is actually only syntactic, for example, there is no global data type definitions in a process and a specification is ended by the reserved word **endspec** instead of **endproc**. The following is a sketch of a complete LOTOS specification.

```
specification Spec [a, b, c ,d] : noexit

type type1 is                    (* begin the global data type definitions *)
 sorts ···
 opns ···
 eqns ···
endtype (* type1 *)

type type2 is
···
endtype (* type2 *)

···
behaviour                        (* begin the behaviour part *)
 P1 [a, b, c]
|[a]|
```

P2 [a, b, d]

where
process P1 [a, b, c] $\cdots$

process P2 [a, b, d] $\cdots$

endspec (* spec *)

2.1.4 Semantics of LOTOS

The operational semantics of LOTOS is defined in terms of Labelled Transition Systems (LTSs). Informally each LOTOS process can be seen as a set of states, with arcs connecting them. These arcs are transitions between the states and are labelled with actions. For basic LOTOS actions are simply the gate names. While for full LOTOS they are pairs consisting of a gate name and a string of data values.

Definition 2.1 (Labelled Transitions System)
An LTS is a quadruple $\langle S, L, T, s0 \rangle$ where S is a set of states, L is a set of observable actions, $T \subseteq S \times (L \cup \{\tau\}) \times S$ is the transition relation, and $s0 \in S$ is the initial state. The class of transition systems with actions in L is denoted by $\mathcal{LTS}(L)$.

A transition in T is also denoted as $s \xrightarrow{\mu} s'$ if $(s, \mu, s') \in T$. The special action $\tau \notin L$ represents an unobservable (or internal) action. In LOTOS *syntax,this unobservable action is named* **i**.

To translate a LOTOS specification to a LTS, LOTOS *inference rules* are applied. The inference rules actually define the meaning of LOTOS operators in terms of LTSs. For example, behaviour *B = i; B'* has the following inference rule, meaning the process *B* can make a transition of **i** then behave like *B'*. Here *B* and *B'* are behaviour expressions.

$$B \xrightarrow{i} B'$$

More complicated inference rules has the form:

$$\frac{P_1, \cdots, P_n}{Q}$$

meaning that given P_1 up to P_n, Q may be derived. For example, the choice operator of LOTOS $B = B_1 [] B_2$ is defined by two rules, where B_1, B_2 and *B* are behaviour expressions and *a* is an action.

$$\frac{B_1 \xrightarrow{a} B'}{B \xrightarrow{a} B'} \quad \frac{B_2 \xrightarrow{a} B'}{B \xrightarrow{a} B'}$$

In the rest of the section, common notations which are employed in the thesis are defined.

Definition 2.2 *Let $p = \langle S, L, T, s0 \rangle$ be an LTS with $s, s' \in S$, let $\mu_i \in L \cup \{\tau\}, a_i \in L$. $L^\star$ denotes the set of all finite action sequences of L and $\sigma \in L^\star$. The following definitions then apply:*

$$
\begin{array}{lll}
s \xrightarrow{\mu_1 \cdot \ldots \cdot \mu_n} s' & =_{def} & \exists s_0, \ldots, s_n : s = s_0 \xrightarrow{\mu_1} s_1 \xrightarrow{\mu_2} \ldots \xrightarrow{\mu_n} s_n = s' \\
s \xrightarrow{\mu_1 \cdot \ldots \cdot \mu_n} & =_{def} & \exists s' : s \xrightarrow{\mu_1 \cdot \ldots \cdot \mu_n} s' \\
s \overset{\mu_1 \cdot \ldots \cdot \mu_n}{\not\longrightarrow} & =_{def} & \textit{not}\ \exists s' : s \xrightarrow{\mu_1 \cdot \ldots \cdot \mu_n} s' \\
s \overset{\epsilon}{\Rightarrow} s' & =_{def} & s = s' \textit{ or } s \xrightarrow{\tau \cdot \ldots \cdot \tau} s' \\
s \overset{a}{\Rightarrow} s' & =_{def} & \exists s_1, s_2 : s \overset{\epsilon}{\Rightarrow} s_1 \xrightarrow{a} s_2 \overset{\epsilon}{\Rightarrow} s' \\
s \overset{a_1 \cdot \ldots \cdot a_n}{\Longrightarrow} s' & =_{def} & \exists s_0 \ldots s_n : s = s_0 \overset{a_1}{\Rightarrow} s_1 \overset{a_2}{\Rightarrow} \ldots \overset{a_n}{\Rightarrow} s_n = s' \\
s \overset{\sigma}{\Rightarrow} & =_{def} & \exists s' : s \overset{\sigma}{\Rightarrow} s' \\
s \overset{\sigma}{\not\Rightarrow} & =_{def} & \textit{not}\ \exists s' : s \overset{\sigma}{\Rightarrow} \\
\textit{init}(p) & =_{def} & \{\mu \in L \cup \{\tau\} \mid p \xrightarrow{\mu}\} \\
\textit{traces}(p) & =_{def} & \{\sigma \in L^\star \mid p \overset{\sigma}{\Rightarrow}\} \\
p \textit{ after } \sigma & =_{def} & \{p' \mid p \overset{\sigma}{\Rightarrow} p'\}
\end{array}
$$

2.2 Overview of DILL

2.2.1 General Consideration

A complete circuit design involves considerations of all requirements such as functionality, timing, power consumption, layout, etc. DILL is designed to address functionality and timing only, which are the most essential aspects that determine the correctness of a circuit and which are specifiable in LOTOS.

Digital logic circuits can be divided into two categories: synchronous circuits and asynchronous circuits. The main difference between them is whether clock signals are employed. Under the control of a global clock signal, synchronous circuits are relatively easy to design so they are the mainstream of today's digital devices. But asynchronous circuits are attracting growing interest because of the potential advantages over their synchronous counterpart. LOTOS is general enough to specify both kinds of circuits, therefore DILL addresses both of them.

The procedure of designing digital circuits can be divided into several steps, as has been shown in figure 1 in chapter 1. Designs at each step can be regarded as a specification at a distinct abstraction level. In most cases, a lower level specification is the structural implementation of the one at the level above it. This requires DILL to support two styles of specifications: behavioural and structural specifications. A *behavioural specification* looks at a system as a black box; it specifies the behaviour of a circuit exhibited on its interface to the environment. Comparatively, a *structural specification* provides the inner structure of a circuit; it specifies how a circuit is built by connecting components.

Each component in a structural specification may also be decomposed into smaller components. But this does not mean components can be decomposed infinitely. In fact the lowest level components specified in DILL are basic logic gates, such as *And*, *Or* and *Inverter* gates etc. In other words, basic logic gates have only behavioural specifications in DILL and cannot be decomposed into, for example, the netlist of transistors.

DILL is intended to be used in real hardware design practice. Therefore it should be easy for design engineers to use it. Because the syntax and semantics of LOTOS are quite different from those of traditional programming languages, many new users find it is difficult to write LOTOS specifications. DILL has a thin layer above LOTOS which makes the specification easier. The layer is written in the m4 macro processing language [Tur94].

Component reuse has been a major theme in software engineering for many years. However, in formal methods there has been little identification of useful specification components using these. A component-based style allows components to be specified and verified individually. Larger combinations of trusted components can then be verified more easily. This architectural view of a system is elaborated in [Tur93], and had a great influence during the development of the DILL project. DILL comes with a large library which contains the specifications of common digital components, such as basic logic gates, flip flops, registers, adders, etc. These specifications have been carefully validated and can be directly used in specifications by referring to the names of the components. Tables 8, 7 and 9 in appendix B summarises the components in the current DILL library.

2.2.2 Underlying Modelling Approach

This section gives the underlying modelling approach of DILL. Many of the models, including those of signals, wires, components. were developed in [TS94]. These models are re-presented here for the completeness of the thesis.

Ports

Every digital circuit has ports through which it accepts inputs and produce outputs. The ports act as the interface of the circuit to its outside environment. DILL abstracts them as LOTOS gates.[1] Normally each LOTOS gate represents a physical port, but it is also possible that a group of ports are modelled as a single LOTOS gate, especially in higher level modelling.

Components and Circuits

A component is a behavioural unit. It could perhaps be modelled as an ADT operation on input values. However, the dynamic behaviour of a logic circuit is often important, so it is better to use LOTOS behaviour

[1] Since `gate' has both a hardware meaning and a LOTOS meaning, the term is qualified when necessary.

expressions. More importantly, the `wiring up' of components specified using ADTs would not be easy. In DILL, components are modelled by LOTOS process declarations that have formal gate parameters for ports. Specific components are then process instances.

Real components have a fan-out (the maximum number of inputs that can be connected to an output). This is a technology restriction that is best ignored in a specification (though a static analysis could determine whether fan-out limits have been complied with). Real components also have a fan-in (the maximum number of outputs that can be connected to an input) that is also technology-dependent.

A circuit can be seen as a special component, th one that is at the highest level of specifications.

Signals

In reality, digital signals take on a range of analogue values (e.g from *0* to *5* volts) but thresholds are set so that they may be treated as logic *0* or *1*. As a signal changes from one value to another, it may pass through an intermediate state that is neither logic *0* nor *1*. It might therefore seem that an `ill-defined' state should be allowed for signals. This, however, is not necessary as an ill-defined signal level should always be transient and therefore should be ignored. As a workable abstraction, signals are regarded strictly as bits.

Logic design proceeds on the basis of binary signals. As an implementation matter there is a choice of how logic *0* and *1* correspond to electrical signals. Normally *0/1* corresponds to low/high, called positive logic. However, negative logic may also be used, with *0/1* corresponding to high/low. This is an implementation decision that depends on the components available. DILL only concerns logic values of signals, thus logic *0* and *1* may correspond to either low or high level in real circuits.

Signals are represented as LOTOS events. There is a choice of whether a continuous signal (a level) or a discrete change in signal (an edge) should be modelled as a LOTOS event. LOTOS, like most specification languages, only deals with discrete events. The initial consideration is that only signal *changes* are modelled. To be a good reflection of real hardware, the direction of a change is explicitly specified by giving the newly established level (e.g. *g!1* for a transition from 0 to 1 on port *g*). As will be discussed in chapter 4, modelling signal *changes* produces difficulties for verifying synchronous circuits. For this kind of circuit, it is assumed that, for each signal, in each clock cycle there is just one stable level and this stable signal level is modelled as a LOTOS event.

In DILL, input signals are usually modelled as events with variable offers while output signals are modelled as events with value offers. For example, *Ip ? newip : Bit* is regarded as an input signal that occurs at input port *Ip*, while *Op ! 1* is an output signal which takes place at port *Op*. In fact, LOTOS does not differentiate inputs and outputs so there are more possibilities in specifications. *Op ? newop : Bit [newop = 1]* is the same output signal as the above, though it is a variable offer that is used.

Wires

Wires or tracks between components are not normally represented explicitly in DILL. In most cases, transmission delays on wires are negligible so representing wires explicitly would unnecessarily complicate a circuit specification. To `wire up' two ports their LOTOS gates are merely synchronized – events at connected ports are matched.

The case where wires are grouped (e.g. a bus) is so common that DILL provides the *MWire* (multi-wire) short-hand notation for this. For example, *MWire(8,D)* represents an 8-bit data bus *D*. However since only the ports of components and not the wires are specified, this really stands for the eight ports *D7, D6, · · ·, D0*.

Bit and BitArray

To represent the values of signals, data types have to be defined. The LOTOS standard data type *Bit* is exploited, with the extension of common logic operations such as *and, or, exclusive or,* etc. A new data type *BitArray* is also provided in the DILL library to represent the signal values on *multi-wires*. Operations of *BitArray* include concatenation, logical functions and comparison functions. A multi-bit signal represented by *BitArray* can also be treated as a set of individual one-bit signals. Further operations could be defined by for specific circuits.

Connecting Components (Structural Specification)

Connecting components is modelled as synchronisation of sub-components at the connected ports. As pointed out in section 2.1.1, LOTOS supports multi-way synchronisation, so it can model the situation where more than two ports are connected. If *Inverter[Ip, Op]* is the LOTOS process modelling an *inverter*, and

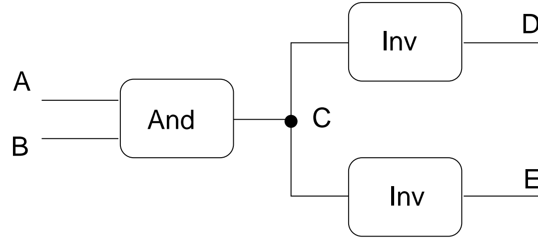


Figure 3: Connecting *And* gates with two *Inverters*

And2[Ip1, Ip2, Op] is the process modelling a 2-input *and* gate, the structure in figure 3 is then specified as:

And2[A, B, C] |[C]| (Inverter[C, D] |[C]| Inverter[C, E])

Note that LOTOS gate *C*, which models the connecting port *C*, synchronizes three processes. This is impossible in other process algebra such as CCS where synchronisation is strictly two-way.

In this circuit, *C* is a port which has no connection with the outside environment. In other words, if the behaviour of the circuit is examined by just looking at its interface (the input and output ports), what happens on *C* is not visible. In LOTOS, such events are abstracted as internal events *i* by hiding them. This helps to define a new process that can be reused by other circuits. For example, figure 3 can be defined as a new 2-input, 2-output *nand* gate.

process Nand2 [A, B, D, E] : **noexit** :=
 hide C **in**
 And2[A, B, C] |[C]| (Inverter[C, D] |[C]| Inverter[C, E])
endproc

If more than one output port of components is connected, the real hardware and its DILL model may behaviour differently. In real hardware, when used properly, connecting several components can implement the logic functions *and* or *or*, depending on the technology used to build the components. This is termed *wire-and* and *wire-or*. But in LOTOS, connecting outputs will almost always result in deadlock. In DILL *wire-and* and *wire-or* have to be transformed into explicit *and* and *or* gates. This does not impose too much restrictions since connecting outputs is often prohibited in digital design to avoid damaging devices.

Multiple Components

MComp (multi-component) is similar to *MWire* and serves as a short-hand for a group of related components. This is useful where a regular structure of identical components is required, as in modelling registers or memories. *MComp* takes a count, a list of ports connecting component instances, and a component definition. The use of arithmetic operators after port names is particularly necessary to ensure that components are connected correctly.

Suppose that the LOTOS process *DFlipFlop[D,C,Q,QBar]* models a D (delay) flip-flop. (Conventionally *D* is the data input, *C* is the clock input, *Q* is the output and *Qbar* is the negated output.) *MComp(4,C=, `DFlipFlop[D,C=,Q,QBar]')* represents a 4-bit register with a clock signal common to each of the flip-flops. (The `=' after *C* means that this port name should be used literally without indexing.) In LOTOS terms, this short-hand stands for:

   DFlipFlop[D3,C,Q3,QBar3]
|[C]|
   DFlipFlop[D2,C,Q2,QBar2]
|[C]|
   DFlipFlop[D1,C,Q1,QBar1]
|[C]|
   DFlipFlop[D0,C,Q0,QBar0]

MComp, together with *MWire* and *BitArray* result in compact descriptions of repeated structures that help DILL to be used in practice.

Behavioural Specification

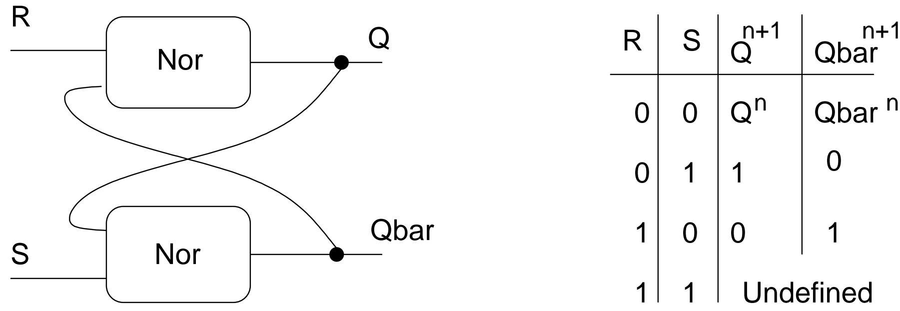


Figure 4: Structure of Reset Set Latch and its truth table

A behavioural specification specifies what a component should do rather than how it is constructed. Writing a behavioural specification of a circuit is comparable to writing an application program using ordinary programming languages. Roughly speaking, there are two ways of specifying behaviour. One takes advantage of ADTs and the other uses behaviour expressions. In the sequel, the former approach is referred to as the *data oriented* style and the later is the *behaviour oriented* style. Either can be used at the specifier's convenience. In the data oriented style, data types are defined for the functions required (e.g. *addition*, *subtraction*, *shift*, etc.), then in the behavioural part of the specification, a *local value definition* (**let** · · · **in**) is used to refer to the data operations. Specifications in the *data oriented* style usually have shorter codes and smaller state spaces, but as can be imagined, defining equations in abstract data types is very difficult for many operations. In the *behaviour oriented* style no extra data is required. Functionality of a circuit is directly specified in the behavioural part of its specification. *Guards* are intensively used to distinguish different states of a circuit, so that the proper values of outputs can be decided. The *behaviour oriented* style may produce longer specifications and larger state spaces, but it can be used for all kinds of circuits. The following gives the fragments of two specification of an RS latch (Reset-Set latch, figure 4). Note that the examples serve as illustration of the two different specification styles and are not necessarily perfect.

The specification in the data oriented style:

```
let newQ:Bit = R nor QBar, newQBar:Bit = S nor Q in
( Q ! newQ; exit ||| Qbar ! newQbar; exit)
...
```

The specification in the behaviour oriented style is as follows:

```
  [(R eq 0) and (S eq 0)] ->
    (Q ! oldQ; exit ||| Qbar ! oldQbar; exit)
[]
  [(R eq 1) and (S eq 0)] ->
    (Q ! 0; exit ||| Qbar ! 1; exit)
[]
  [(R eq 0) and (S eq 1)] ->
    (Q ! 1; exit ||| Qbar ! 0; exit)
[]
  [(R eq 1) and (S eq 1)] ->
      Error; ...
```

2.3 Conclusion

This chapter introduced LOTOS and gave an overview of the DILL approach. General considerations in DILL were explained, together with the underlying modelling approach adopted in DILL. It can be seen that LOTOS can be used to model digital circuits in a very natural manner, due to the clear correspondences between the concepts in LOTOS and the elements in digital circuits. For example there are close

correspondences between LOTOS gates and circuit ports, LOTOS events and digital signals, processes and components. For the illustration purpose, the circuit specified in this and the following chapters are relatively small. But this does not mean that DILL can only cope with small circuits. Some components in the DILL library (see appendix B) are much larger than those illustrated in the thesis. In [JT97], there is also a case study of specifying a CPU in both behavioural and structural styles. The CPU is made up of several sub-parts including instruction decoder, ALU (Arithmetic and Logic Unit) and registers. This case study reveals DILL's capability of dealing with large circuits.

3 Specification of Digital Logic Circuits

This chapter describes DILL models of digital components in synchronous and asynchronous circuits. Background knowledge is first introduced, including the differences between the two kinds of circuits, some potential advantages of asynchronous circuits, and the delay and environment models. It is followed by a presentation of the model of basic logic gates developed in [TS94], which was the first component model developed in the DILL approach. The chapter then focuses on how to specify synchronous and asynchronous circuits respectively. Synchronous circuits are typically made up of combinational logic components and storage elements, with the basic logic gates and D flip flop being the representatives of them. It is discovered that the model of basic logic gates in [TS94] is not suitable to be used in modelling synchronous circuits. A new model is therefore designed. The model for storage elements are also given in the chapter. These two models make it possible to specify any synchronous circuits in the structural style. The chapter also discusses the guildlines for specifying the behaviour of synchronous circuits. For specifying asynchronous circuits, different classes of asynchronous circuits are first introduced. Since LOTOS abstracts away timing characteristics of systems being specified, only those classes which assume unbounded delay models are addresses in this chapter. Basic logic gates are again employed as the illustrative examples. Towards the end of the chapter, it is revealed that when LOTOS events model signal transitions in digital circuits, there is a gap between the behaviour of LOTOS specifications and the behaviour of real circuits. As a result, LOTOS specifications represent only part of possible behaviour that the real circuits may exhibited. A solution for the problem is proposed by introducing input quasi-receptive specifications. Throughout the chapter examples and case studies are presented to illustrate the approach.

3.1 Background

3.1.1 Synchronous Circuits and Asynchronous Circuits

Before defining synchronous and asynchronous circuits, the concepts of combinational circuits and sequential circuits are required. A circuit whose outputs are purely determined by its current inputs is termed a *combinational* circuit. If outputs are decided not only by current inputs but also previous inputs, it is a *sequential* circuit. Sequential circuits contain *storage elements* such as latches or flip flops to remember information related to previous inputs. This is mainly achieved by feeding back outputs to inputs. The RS (Reset-Set) latch in figure 4 is one such storage element.

Basically, two kinds of sequential circuits exist. In a *synchronous sequential circuit* (*synchronous circuit* for short), there is a global clock which controls all storage elements in the circuit. Only when particular points of the clock cycle come can storage elements change their states and consequently cause changes of outputs. Input changes before such points cannot directly influence states of a circuit. An *asynchronous sequential circuit* (or *asynchronous circuit* for short), on the other hand, does not have a global clock, so any new input may result in changes of states and outputs. *Clocked* and *unclocked* circuits are alternative names for these two kinds of circuits to clearly reflect their main difference.

But this difference is not always evident since some circuits combine both features. It is not rare that a circuit can be controlled by more than one clock signal. As a result its storage elements change its states at different times. This is normally regarded as a synchronous circuit. But in this thesis, only those circuits with one global clock are considered. Some other circuits may have several *local clocks* controlling several parts of the circuits, but other parts and their connections are still asynchronous. Again this kind of circuit is not in the scope of the thesis.

The mainstream of today's digital device is synchronous. Controlled by a global clock signal, the behaviour of a synchronous circuit is actually discrete: the circuit is assumed to have a finite number of states, and after one or more time units, it changes its state from one to the other. This effectively filters out the influence of transient signal transitions between two time instants. As a consequence, designing synchronous circuits is substantially simplified since there is no need to consider *hazards*, the transient signal transitions which are caused by the propagation delay of digital components.

Asynchronous circuit design, on the other hand, is more complicated because hazards have to be completely eliminated before a design is completed. This is unfortunately a very hard task and prevents asynchronous circuits from being widely used. Nevertheless, there has been a resurgence of interest in asyn-

chronous design methodology recently due to several potential benefits of removing global clocks. Among others, the following are some of the benefits of asynchronous circuits [DN95]:

Absence of clock skew: The difference in arrival times of a clock signal at different parts of a circuit is referred to as *clock skew*. Typically, clock skew is accommodated by longer clock cycles, which results in reduced maximum clock frequency. As VLSI systems become smaller, denser and faster, clock skew becomes increasingly severe and deskewing becomes harder and more expensive [DN95]. Asynchronous circuits get rid of this problem by eliminating the concept of global clocks.

Potential for low power: Power consumption is a major concern in the markets for portable equipment, where battery life is crucial. In synchronous systems, the global clock toggles clock lines, charging and discharging capacitance throughout the system, even in portions unused in current computations. In asynchronous systems, circuit components are activated only when necessary, and remain idle at other times without dissipating significant power.

Potential for high performance: Synchronous circuits have to be designed for worst-case conditions because clock cycles are adjusted according to the slowest operations that might be required, even though in most cases operations complete in much shorter time than the worst case. Asynchronous systems can be optimized for the average-case conditions, with each operation taking as long as required for any particular situation.

Better technology migration potential: Asynchronous design approaches allow a system to be designed as a set of sub-systems communicating via interfaces. Since there is no global synchronization, components in an asynchronous circuit can be easily substituted by faster ones (as long as interfaces are compatible), without changing functionalities of the original ones but improving the performance dramatically. By contrast, in synchronous systems, overall performance depends on worst-case conditions and therefore it is often the case that in order to improve the speed potential of a new technology, reorganization of the whole system is required to deal with new worst-case conditions.

The recent active study of asynchronous circuits has resulted in very large scale designs, including asynchronous processors such as AMULET [FPJ+94] by Manchester University, Counterflow pipeline processor[SSM94] by SUN Labs, TITAC [NUK+94] by Tokyo Institute of Technology, and STRiP [Dea92] by Stanford University.

3.1.2 Delay and Environmental Models

Delay models are the abstractions of delay characteristics of components comprising a circuit. A component has *bounded delay* if an upper and lower bound for the delay magnitude is known. Otherwise it has *unbounded delay*, which means no bound is known except that it is finite. The unbounded delay model is more robust than the bounded one. That is to say, a circuit designed under the assumption of unbounded delay can usually work correctly when the actual delay model is bounded, but not vice versa.

Delays can also be characterized as *pure* or *inertial* [Ung69]. Suppose the delay of a digital component is D. If a component has pure delay, all input changes will have an effect on output. In other words, outputs follows inputs after delay D. If the component has inertial delay, output will respond only to input changes which have persisted for time D. As a result, input pulses whose width is less than D will be absorbed by the component. This reflects the fact that short pulses contain insufficient energy to trigger a state change in a real component. Figure 5 gives these two basic delay models, as can be seen, pure delay does not alter a waveform, while inertial delay may do so by eliminating short glitches, i.e. the narrow pulses in a waveform.

A useful digital circuit should inevitably have interactions with its outside world. A circuit and its environment forms a closed system, called a *complete circuit*. If the environment must wait for a circuit to stabilize before providing new inputs to the circuit, the two interact in *fundamental mode* [Ung69]. Otherwise the interaction is termed *input/output mode*, meaning that the stability of a circuit is not required before it is allowed to receive further inputs. Input/output mode is more robust than fundamental mode.

The concepts of delay and environment model are especially important for asynchronous circuits, because an asynchronous circuit designed with certain delay and environment assumptions normally cannot

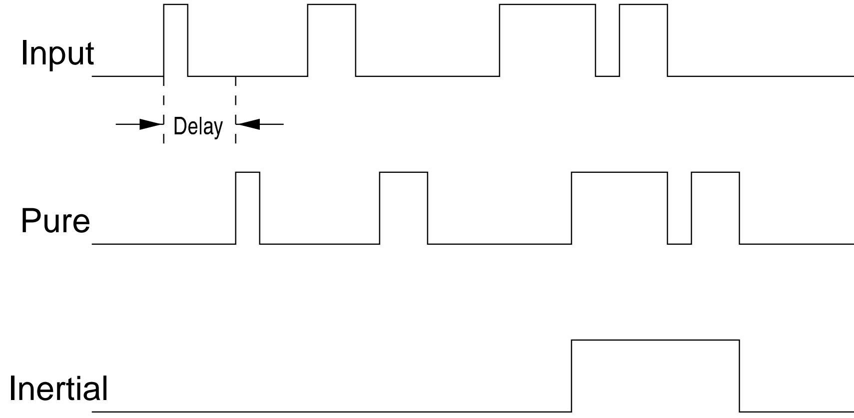


Figure 5: Inertial and pure delay

work correctly with others. In contrast, synchronous circuits tend to have more unified delay and environment models. Because clock cycles are calculated according to the maximal delay of components, all synchronous circuits are actually based on the bounded delay assumption. The environment of synchronous circuits can be regarded as for the fundamental mode since the effects of inputs before stabilization of circuits are filtered out or delayed by means of clock signals.

3.2 The First Model of Basic Logic Gates

Specifying basic logic gates is very important in DILL not only because they are the most basic building blocks of both synchronous and asynchronous circuits, but also because they are representative. Behavioural specifications of other higher-level components follow the same specification method.

In this section, the model of basic logic gates developed in [TS94] was presented and examined. This is the first component model developed in DILL approach. The initial idea was to have a common model that could be used in both synchronous and asynchronous circuits. Although this appears straightforward, the model was obtained after considerable thought. The main difficulty is that since the model is intended to be general, little can be assumed about the environment. The resultant model is a faithful representation of logic gates in the real world. The following takes as example a 2-input gate *Nand2*. Specifications of other logic gates are almost identical except that different logic operators are used in the **let** expression. A *Nand2* gate with both initial inputs of *0* and output of *1* can be instantiated as *Nand2 [Ip1, Ip2, Op] (0, 0, 1)*.

```
process Nand2 [Ip1, Ip2, Op] : (dtIp1, dtIp2, dtOp : Bit) noexit :=
 Ip1 ? newdtIp1 : Bit [newdtIp1 ne dtIp1];                    (* one input is changed*)
 Nand2 [Ip1, Ip2, Op] (newdtIp1, dtIp2, dtOp)                   (* repeat behaviour *)
[]
 Ip2 ?newdtIp2 : Bit [newdtIp2 ne dtIp2]                   (* other input is changed *)
 Nand2 [Ip1, Ip2, Op] (dtIp1, newdtIp2, dtOp]                   (* repeat behaviour *)
[]
 let newdtOp : Bit = Apply (Nand, dtIp1, dtIp2) in             (* new Output *)
   [newdtOp ne dtOp] -> Op ! newdtOp;                             (* Output Change *)
   Nand2 [Ip1, Ip2, Op] (dtIp1, dtIp2, newdtOp)                 (* repeat behaviour *)
endproc (* Nand2 *)
```

There are several key points implied in this process:

In lines 2, 5, and 8, events occur only when they have value changes (achieved by LOTOS selection predicates and guards). This implies that LOTOS events model signal transitions. LOTOS events can only deal with discrete actions. Modelling continuous signal levels would result in infinite events in a finite period.

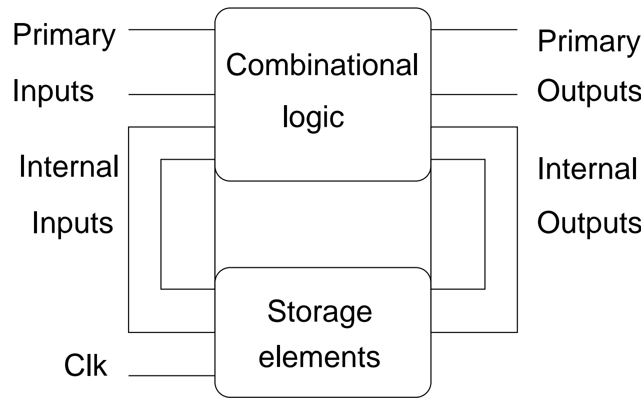


Figure 6: Structure of synchronous circuits

Because the three sub-behaviour expressions in the process are combined by the choice operator **[]**, inputs and output can occur at any time as long as there are value changes. That inputs are *always* allowed to change is termed *input receptiveness* in some literature [Dil89]. As will seen in section 3.4.5, receptiveness is very important for correctly modelling components in asynchronous circuits.

In this model, an input change may pre-empt a pending output. For example, if initially *Ip1, Ip2, Op* are *0, 1, 1*. After *Ip1* changes from *0* to *1*, *Op* can change to *0*. However, if *Ip2* changes to *0* before *Op* changes, the newest value on *Op* will still be *1*. In other words, the potential *1* to *0* change on *Op* is pre-empted.

Pre-empting potential output indicates that an input change comes earlier than the propagation delay allows. This actually follows the assumption of the inertial delay model discussed in section 3.1.2.

In short, the first model of basic logic gates models signal transitions as LOTOS events. It is specified in the inputs receptive manner, and assumes the components have inertial delay model, which implies that pending outputs may be pre-empted.

3.3 Specifying Synchronous Circuits

3.3.1 Structure of Synchronous Circuits

The general structure of a synchronous circuit is shown in Figure 6. It is made up of two parts: *combinational logic* and *storage elements*. The former part does logic calculation, and the latter stores states of a circuit. Combinational logic provides the primary outputs and internal outputs according to the primary inputs and internal inputs. Internal outputs are then fed into storage elements to produce internal inputs. Since these storage elements are controlled by a clock signal, changes of the internal inputs are synchronised with the clock, in other words they are changed only at particular moments of the clock cycle (usually its transitions). This allows internal outputs to settle down, filtering out transient signal transitions caused by propagation delays of the combinational logic. Consequently, primary outputs are not influenced by these transient signals either.

It is easy to see that for a synchronous circuit operating correctly, designers must ensure that the clock cycle is slower than the slowest combinational logic so that the whole circuit can settle down before it changes its state. This can be done by analysing timing characteristics of the components used in circuits. The untimed version of DILL cannot of course confirm if this clock constraint is met or not. However as discussed elsewhere (chapter 6), timed LOTOS can specify such constraints. Instead, sections 3.3.3 and 3.3.5 will show that properly modelling storage components and the environment can ensure that synchronous circuits specified in DILL fulfill the clock condition automatically.

In the practice of synchronous design, primary inputs are usually synchronised with a clock signal. This makes designing and analysing synchronous circuits much easier. DILL incorporates this practice into its synchronous circuit model, assuming that primary inputs have already been synchronised with the clock signal.

Apart from this, the DILL synchronous model has two more restrictions. It is important that there

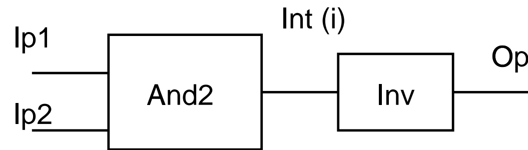


Figure 7: An implementation of a *Nand2* gate

is no cyclic connection within the combinational logic, and storage elements have to be specified in a behavioural style. These restrictions are related to the way components are modelled, for otherwise a DILL specification might deadlock while a real circuit could still work. This will be discussed further in sections 3.3.2 and 3.3.3.

3.3.2 Basic Logic Gates in Synchronous Circuits

In this section, the DILL specification of basic logic gates is re-investigated in the context of synchronous circuits. Although it is a good representation of real-world logic gates, the model discussed in section 3.2 makes it very difficult to analyse the behaviour of synchronous circuits.

Suppose the example in figure 7 is a combinational stage of a certain circuit. Initially *Ip1, Ip2, Op* are *0, 0, 1*. After the input sequence *Ip1!1, Ip2!1, Ip1!0*, *Op* may either remain at *1* or change to *0* then back to *1*, which depends on whether *Ip1!0* comes before or after the output *int* of the *And2* gate, i.e. whether the change on *Ip1* is faster or slower than the propagation delay of the *And2* gate. In the first case, *Ip1* is a fast input change so the pending output is pre-empted, consequently *Op* stays at 1. While in the second case, the output of the *And2* gate occurs before *Ip1!0*, so it is possible for the *Inverter* to change to *0* then back to *1*. If the clock cycle of a synchronous circuit is slow enough to allow the *Nand2* gate settle down, the *1, 0, 1* is only a temporary transition. The two different behaviour does not necessarily to be distinguished since only the settled signal level can influence the behaviour of the whole circuit. The problematic thing is that there appears no way to 'sense' when the combinational logic has settled down, which makes automatic analysis of circuits almost impossible. This suggests that a new model of basic logic gates is needed which takes the characteristics of synchronous circuits into account.

As discussed, in each clock cycle only the settled level of each signal is of interest. Consider figure 6 again. Suppose that there is an environment which offers each primary input an event once and only once within a clock cycle. (This is reasonable because DILL assumes that the primary inputs are synchronised with the clock.) Suppose further that storage elements produce an output once in each clock cycle either (see section 3.3.3). Under this condition, if a basic logic gate is modelled in such a way that output events happen only after all inputs occur, then each output event happens exactly once as well. Moreover if input events model settled signals, so do the output events. In this way, transient signal transitions resulting from different arrival times of different input events can be filtered out.

Note that this model requires each signal to appear once in a clock cycle. In other words, no matter if the value of this signal changes or not, there should be an event offer in the corresponding clock cycle. LOTOS events thus no longer model signal transitions on wires, but rather signal levels. For instance, the LOTOS event *Ip!0* means that in a certain clock cycle the signal level on wire *Ip* is *0*. (A similar argument applies for *Ip!1*). The level on the same wire during the previous cycle could be *0* or *1*, but the event itself does not give any information about its previous level.

Following the way that basic logic gates are modelled, every wire in a synchronous circuit has just one associated event offer during a clock cycle. This answers why there is no need to worry about the infiniteness resulting from modelling LOTOS events as signal levels. Usually if an event represents a signal level, there will be an infinite number of events during an arbitrary time interval because the level is a continuous variable. However for the case of synchronous circuits, whose progress is actually in discrete steps, settled signal levels constitute discrete variables.

To illustrate the above idea, a re-specification of *Nand2* gate is given below. Note that inputs are interleaved, i.e. they can occur in any order. It might appear that the order of input events could be fixed since it does not influence the functionality of a component. This would result in a smaller state space

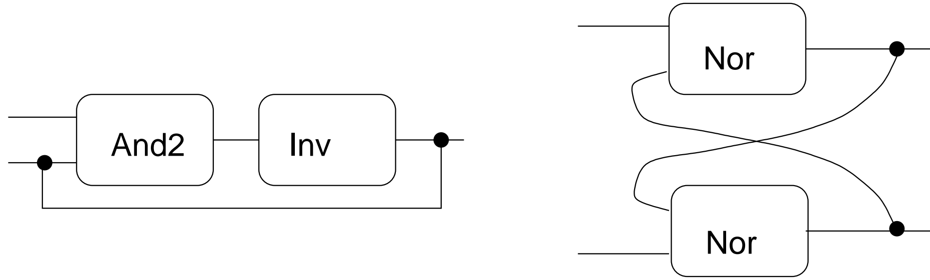


Figure 8: Examples of cyclic connections

when circuits are verified. Unfortunately this could cause deadlock when components are connected. For example, suppose that components *A* and *B* each have two inputs. Imagine that the fixed order of inputs is *IpA1* before *IpA2*, and *IpB1* before *IpB2*. This would lead to deadlock if the components share inputs, with *IpA1* connected to *IpB2* and *IpA2* connected to *IpB1*. For this reason, DILL insists on fully interleaved inputs.

```
process Nand2 [Ip1, Ip2, Op] : noexit :=
 ( Ip1 ?dtIp1 : Bit; exit (dtIp1, any Bit)                    (* allow one input *)
 |||
   Ip2 ?dtIp2 : Bit; exit (any Bit, dtIp2))                   (* allow other input *)
 ≫ accept dtIp1, dtIp2 : Bit in                               (* accept both inputs *)
 ( Op !(dtIP1 nand dtIp2);                                    (* output nand of inputs *)
   Nand2 [Ip1, Ip2, Op])                                      (* repeat behaviour *)
endproc (* Nand2 *)
```

In contrast to the specification in section 3.2, the above process does not concern delay aspects of logic gates. A signal propagating through an inertial delay component or a pure delay one can have only the same stable level, and only this stable level is of interest in synchronous circuits. Details of the delay types are therefore better to be abstracted away. Another difference is that inputs are no longer receptive. This appears unrealistic, but in the context of synchronous circuits, every input has only one stable level during a clock cycle, so it is not necessary to make it receptive.

Because inputs and outputs alternate in this model, there should be no cyclic connection within a combinational stage, for otherwise a DILL specification would deadlock. This arises because feedback connections make inputs and outputs dependent on each other. Figure 8 gives examples of such cyclic connections, with the right hand one being a common building block of latches and flip-flops. This is why storage elements cannot be specified in the structural style.

3.3.3 Specifying Storage Elements

A storage element can store one logic value (*1* or *0*). The stored value is decided by the inputs of the element during the effective instant of a clock cycle (for example, a positive transition of the clock) and remains unchanged until the next effective instant comes. Temporary input changes between these two effective instants , which are regarded as hazards, do not have effects on the value held in storage elements.

Storage elements are modelled in the behavioural style. The following takes as example one of the simpler storage elements: a D flip flop (DFF). A DFF (Delay Flip-Flop) has input *D*, clock input *Clk* and output *Q* (some may also have the inverted output). Here the DFF is assumed to be *positive edge triggered*, which means that output *Q* changes to the same level as *D* after the *Clk* changes from *0* to *1*. Unlike the specification of basic logic gates, storage elements have states associated with them, which is reflected in value parameters *dtD*.

```
process DFF [D, Clk, Q] (dtD, dtClk : Bit) : noexit :=
  D ? newdtD : Bit; DFF [D, Clk, Q] (newdtD, dtClk)           (* input new data *)
[]
  Clk ? newdtClk : Bit;                                       (* input clock pulse *)
```

```
    ( [(dtClk eq 1) and (newdtClk eq 0)] ->                    (* ignore -ve pulse *)
        DFF [D, Clk, Q] (dtD, newdtClk)                         (* continue behaviour *)
    []
      [(dtClk eq 0) and (newdtClk eq 1)] ->                     (* react to +ve pulse *)
        Q ! dtD;                                                (* output stored data *)
        DFF [D, Clk, Q] (dtD, newdtClk))                        (* continue behaviour *)
endproc (* DFF *)
```

Suppose there is an internal output feeding into this flip-flop. If the clock signal is not constrained, it is possible that the clock moves to the next cycle before the combinational logic has settled down. The model of a synchronous circuit must exclude this possibility. After a positive-going transition of the clock signal, if the *D* input of the flip-flop has not occurred yet then the next positive-going transition of clock signal must not occur. This is ensured by the following constraint on the D flip-flop specification. Process *Cons_DFF* deals with the initial state of the flip-flop, where the *D* input and *Clk* input can occur in any order. However, after the first positive edge of the *Clk* signal, the next positive edge has to wait for the *D* signal to ensure its previous combinational stage has settle. This is specified in process *Cons_DFF_Aux*. The return of *Clk* to *0* is allowed either before or after the *D* event. Thus there are two possibilities in *Cons_DFF_Aux*. It is attractive to specify *Cons_DFF* in the same way as *Cons_DFF_Aux*, i.e. it appears reasonable to assume that the clock arrives after the data input *D* has settled down from the initial state. However, this will result in specification deadlock when two *DFFs* are connected in series. In other words, when the *Q* output of the first *DFF* is the *D* input of the second. Suppose this shared port is called *QD*, then QD should wait until *Clk* has happened for the first *DFF*, while it should happen before *Clk* for the second one. Deadlock is therefore inevitable. The full specification of a D flip-flop combines *DFF* and *Cons_DFF*, i.e. *DFF* | [*D, Clk*] | *Cons_DFF [D, Clk]*.

```
process Cons_DFF [D, Clk] (dtClk : Bit) : noexit :=
    D ?newdtD : Bit;                                            (* input new data *)
    Cons_DFF [D, Clk] (dtClk)                                   (* continue behaviour *)
[]
    Clk ?newdtClk : Bit;                                        (* input clock pulse *)
    ( [(newdtClk eq 1) and (dtClk eq 0)] ->                     (* react to +ve pulse *)
        Cons_DFF_Aux [D, Clk] (newdtClk)                        (* after one clock pulse *)
    []    [(newdtClk ne 1) or (dtclk ne 0)]->                   (* ignore other pulses *)
        Cons_DFF [D, Clk] (newdtClk))                           (* continue behaviour *)
  where
    process Cons_DFF_Aux [D, Clk] (dtClk : Bit) : noexit :=
        D ?newdtD : Bit; Clk !0; Clk !1;                        (* input before -ve pulse *)
        Cons_DFF_Aux [D, Clk] (1)                               (* continue behaviour *)
    []
        Clk !0; D ?newdtD : Bit; Clk !1;                        (* input after -ve pulse *)
        Cons_DFF_Aux [D, Clk] (1)                               (* continue behaviour *)
    endproc (* Cons_DFF_Aux *)
endproc (* Cons_DFF *)
```

3.3.4 Specifying Circuit Behaviour

Specifying behaviour of a whole circuit uses a clock cycle-by-cycle basis. In each clock cycle, output behaviour is specified according to inputs and internal states. Essentially a synchronous circuit is a storage element, but may have more complicated logic and more internal states. Clock signals can be implicit at the highest level of specification because no connection is required at this level. In fact, it is found that it is more convenient to make clock signals implicit during high-level specification. Moreover, smaller state spaces result due to implicit clocks.

3.3.5 Case Study: Specifying a Single Pulser

In this section, a small synchronous circuit called the *single pulser* is specified in behavioural and structural styles. The single pulser is a standard hardware verification benchmark documented in [SK96]. The simple behaviour and small size of its design make it a good example for illustrating the DILL approach.

A Single Pulser is a clocked-sequential device with a one-bit input *P_In* and a one-bit output *P_Out*. It deals with a debounced switch that is on (true) in the down position and off (false) in the up position. The goal is to devise a circuit to sense the switch being turned on, asserting an output signal lasting one clock cycle. The system should not allow additional outputs until the user has turned the switch off.

The description does not make clear when the output pulse should be asserted: on pressing the switch (*P_In* from *false* to *true*), or releasing the switch (*P_In* from *true* to *false*)? For convenience, the first case is termed *positive triggered* and the other one is *negative triggered*.

In the following, the input *P_In* is assumed initially in the off position and the clock signal is implicit. In each clock cycle, if there is an active edge [2]on signal *P_In*, *P_Out* is asserted. Otherwise, if *P_In* does not change or is not going an active edge, *P_Out* should be *0*. This ensures *P_Out* is asserted only at active edges of *P_In*, and lasts for just one clock cycle.

```
process SP [Ip, Op] : noexit :=                          (* Single Pulser *)
  i; SP_P [Ip, Op] (0)                    (* +ve triggered implementation *)
[]
  i; SP_N [Ip, Op] (0)                     (*-ve triggered implementation *)
where
  process SP_P [Ip, Op] (dtI: Bit) : noexit :=
    Ip ?newI : Bit;                                     (* get new input *)
    ( Op !1 [(dtI eq 0) and (newI eq 1)];         (* output 1 on 0->1 input *)
      SP_P [Ip, Op] (newI)
    []
      Op !0 [not ((dtI eq 0) and (newI eq 1))];          (* else output 0 *)
      SP_P [Ip, Op] (newI) )
  endproc (* SP_P *)
  process SP_N [Ip, Op] (dtI: Bit) : noexit :=
    Ip ?newI : Bit;                                     (* get new input *)
    ( Op !1 [(dtI eq 1) and (newI eq 0)];         (* output 1 on 1->0 input *)
      SP_N [Ip, Op] (newI)
    []
      Op !0 [not ((dtI eq 1) and (newI eq 0))];          (* else output 0 *)
      SP_N [Ip, Op] (newI) )
  endproc (* SP_N *)
endproc (* SP *)
```

Figure 9 shows a design for the single pulser that is given in the benchmark. The clock is hidden in the structural specification:

```
hide Inp, N_Find, Find, Clk in
  ((DFF |[N_Find, Inp]| (Inverter |[Find]| And2)) |[Clk, Inp]| DFF)
|[P_In, Clk, P_Out]|
Env [P_In, Clk, P_Out]
```

The *Env* process serves as the environmental constraint on the circuit. It permits *P_In* to come before each positive-going clock transition, and allows the next clock cycle to come only after *P_Out* has occurred. The constraint between *P_In* and *Clk* ensures that *P_In* is synchronised with *Clk*, and the constraint between *Clk* and output respects the slow-clock requirement: *P_Out* must settle down before the next positive going clock transition. These assumptions are not automatically guaranteed by the circuit specification, but they are required by the DILL synchronous circuit model. In outline, *Env* is specified as follows:

[2]For positive triggered single pulser, the active edge is the positive edge of the *P_In*, it is the negative edge of *P_In* if the single pulser is negative triggered.

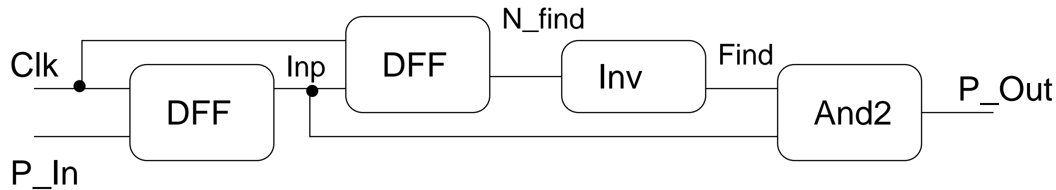


Figure 9: Single pulser design

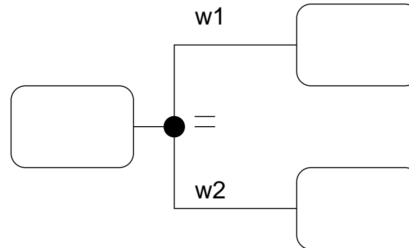


Figure 10: An isochronic fork

```
process Env [P_In, Clk, P_Out] : noexit :=
(P_In ?dtPIn : Bit;
 Clk !1;
 (Clk !0; exit ||| P_Out ?dtPOut : Bit; exit)
) ≫ Env [P_In, Clk, P_Out]
endproc
```

3.4 Specifying Asynchronous Circuits

3.4.1 Classes of Asynchronous Circuits

Unlike synchronous circuits, which have a unified structure, asynchronous circuits exhibit a variety of forms due to the different delay and environment assumptions made. An asynchronous circuit can only behave correctly when these assumptions are met.

Delay insensitive circuits (DI): DI circuits [Udd86] are the most robust class in the asynchronous circuit family since they take the most pessimistic view about delays and the environment. Delays on both components and wires are assumed to be unbounded, and the environment is in input/output mode (see section 3.1.2). DI circuits can operate correctly regardless of delay magnitudes on wires and components, as long as they are finite. Martin [Mar90] has proved that the class of purely DI circuits designed using single-output gates is limited, which means that most meaningful DI circuits cannot just be built purely from basic logic gates. Some special components [MFR85] are therefore designed.

Quasi delay insensitive circuits (QDI): QDI circuits augment the delay model of DI circuits with the *isochronic forks* assumption [Mar90]. *Isochronic forks* are forking wires on which the difference of delay magnitudes is negligible, as shown in figure 10 where delays on *w1* and *w2* are regarded as equal. This seems the weakest compromise to pure DI circuits to build practical circuits using single-output gates [Mar90]. QDI circuits assume input/output mode environment.

Speed-independent circuit (SI): Design of SI circuits was pioneered by Muller [MB59]. In this class of circuits, gates are assumed to have unbounded delay while wires have zero delay. If all gates have just one output, SI and QDI are actually identical, see section 3.4.2 for more detail. SI circuits assume input/output mode environment.

Fundamental mode Huffman circuits: A Huffman sequential circuit [Ung69] can be modelled as a combinational logic block with a subset of outputs feeding back to inputs to hold states. Gates and wires are assumed to have bounded delay, and the environment is in fundamental mode. In its most basic form, only one input is allowed to change each time, and the next input change has to wait until the circuit is stable. Obviously this kind of circuit is not so useful in real designs as the speed of circuits can be very slow.

Burst mode circuits: Burst-mode circuits [DCS93]) are an extension of fundamental Huffman circuits. They allow one or more input bursts to occur at each state. Inputs in a burst may occur in any order, and the circuit does not react until the entire input burst has finished. The next input burst can come only after the specified output burst has completed, and the circuit has been stable. In fact burst-mode systems still require the fundamental mode assumption, but only between different input bursts.

Micropipeline: Ivan Sutherland introduced the concept of *micropipeline* in his 1988 Turing Award lecture [Sut89]. A micropipeline can be divided into two parts: a control part which assumes an unbounded delay model and thus could be implemented, for example, in DI circuits; and a datapath part which adopts a bounded delay model. Between two stages of a micropipeline, a *bundled data protocol* is applied: the delay on data wires must be less than that on control wires so that stable data is transfered from one stage to the other before the corresponding control signals occur.

Specifying bounded delay needs a formalism which supports quantitative timing specification. This chapter mainly studies those classes assuming unbounded delays, namely the DI, QDI and SI circuits. Chapter 6 deals with specifying bounded delays using ET-LOTOS.

3.4.2 DILL and Speed Independent Circuits

Among different classes of asynchronous circuits, speed independent circuits match the modelling techniques of DILL most closely: in speed independent circuits, propagation delays of components are unbounded. In LOTOS the interval between the occurrence of two concatenated events is also unbounded. In DILL wiring up two ports is done by synchronising the LOTOS events, which actually assumes that delay on the connecting wires is negligible, an assumption which is also adopted by SI circuits.

The other circuits with unbounded delay models, namely DI (delay insensitive) and QDI (quasi-delay-insensitive), can be easily changed to SI circuits by inserting artificial delay components. In figure 11, a DI circuit(figure 11 (A)) can be remodelled as an SI circuit (figure 11(B)) by inserting artificial delay components on each wire. Note that in the figure lowercase letters represent delays on wires or components. Actually, as unbounded delay plus unbounded delay is still unbounded, so most of the wire delays can be accumulated with their preceding components. Only forks and components with more than one outputs should be otherwise treated, as shown in figure 11(C). Figure 11(D) is the SI representation of figure 11(A) when it is regarded as a quasi delay-insensitive circuits. Since $h = i$ and $k = l$, only wires from the components with multi-outputs are inserted with additional delay components. Figure 11(D) also shows that if every component has a single output, QDI and SI are identical.

Speed independence is closely related to the concept of *semi-modularity*. Under the delay model of speed independent circuits, if no components in a circuit can ever receive an input which can change the level of pending outputs, the circuit is termed *semi-modular* [BM91, BZ97]. For instance, suppose a two-input *And2* gate has a pending output *1*. If it receives a *0* input on one of its inputs before the output 1 is produced, the *And2* gate is not semi-modular since this *0* input might change the pending output from *1* to *0*. Non-semi-modularity indicates that at least one component in a circuit has speed dependent behaviour. In the above example, after receiving the input *0*, the output of the *And2* gate depends on its speed: a faster gate can produce 1 followed by 0, while a slower one can only produce 0. Semi-modularity is usually regarded as a basic characteristic of speed independent circuits [BBM94, KKTV94].

3.4.3 Basic Logic Gates in SI Circuits

Unlike the case of modelling synchronous circuits, modelling asynchronous circuits requires that LOTOS events represent signal transitions since every transition may influence the behaviour of circuits.

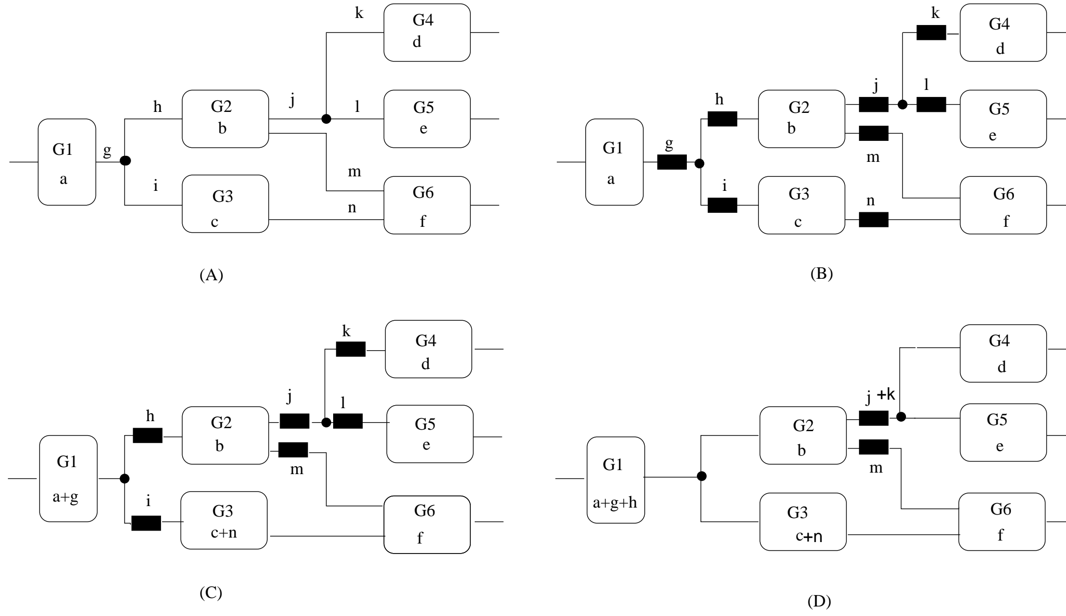


Figure 11: Modelling DI, QDI as SI

The specification in section 3.2 is one of the possible models of the basic logic gates in SI circuits. It is only suitable for those gates exhibiting inertial delay, since it allows new inputs to pre-empt pending outputs. If inputs which may result in the pre-emption are prohibited, it becomes a model which satisfies the requirement of semi-modularity. Take the example of the *Nand2* gate. Suppose its inputs and output *Ip1, Ip2, Op* are initially *1, 1, 0*. After *Ip1* changes to *0*, its output should change to *1* accordingly. If, before the output happens, *Ip1* changes back to *1*, then the new output will eventually be *0*. The model in section 3.2 allows the change on *Ip1*, resulting in speed dependent behaviour. If this input is not allowed, a new model of basic logic gates which respects semi-modularity can then be obtained:

```
process Nand2[Ip1, Ip2, Op] ( dtIp1, dtIp2, dtOp : Bit) : noexit :=
 let newOut : Bit = dtIp1 nand dtIp2 in                    (* potential output *)
  (Ip1 ? new1 : Bit [(new1 ne dtIp1) and                   (* signal transition *)
                     ((dtOp eq newOut) or               (* no new potential output *)
                     ((dtOp ne newOut) and           (* there is potential outputs *)
                     ((new1 nand dtIp2) eq newOut)))];     (* but it won' t be changed *)
  Nand2[Ip1, Ip2, Op] (new1, dtIp2, dtOp)                  (* continue behaviour *)
 []
  Ip2 ? new2 : Bit [(new2 ne dtIp2) and                    (* signal transition *)
                     ((dtOp eq newOut) or               (* no new potential output *)
                     ((dtOp ne newOut) and            (* there is potential output *)
                     ((new2 nand new1) eq newOut)))];      (* but it won' t be changed *)
  Nand2[Ip1, Ip2, Op] (dtIp1, new2, dtOp)                  (* continue behaviour *)
 []
  Op ! newOut [dtOp ne newOut];                            (* new output produced *)
  Nand2[Ip1, Ip2, Op] (dtIp1, dtIp2, newOut))              (* continue behaviour *)
endproc
```

Compared to the specification in section 3.2, the only difference is the selection predicates behind input events. Here the constraint of semi-modularity is required. An input offer can only happen when there is no potential output, or even though there is such output, the new input will not alter it ((new1 nand dtIp2) eq newOut).

A question is raised as there are two models of basic logic gates for SI circuits: which one is better? The one in section 3.2 is a full specification of logic gates in the sense that it specifies the behaviour under

all possible input situations (i.e. it is input receptive). But it requires that the specified hardware has inertial delay characteristics. When the assumption cannot be guaranteed, the model may not be a suitable one. As for the model above, it is a partial specification in the sense that it does not allow certain input events to happen at some times. More precisely, those inputs which may alter potential outputs are prohibited. This is the stricter model of the two, and can be used for checking if a circuit is really semi-modular or not. For some components, such as the other basic building blocks of SI which will be introduced in the next section, full specifications are not available since these components are not yet as standard as basic logic gates. Different implementations of the components may have different behaviour under the unexpected inputs, in which case partial specifications are the only choice.

3.4.4 Other Basic Building Blocks of SI Circuits

Besides basic logic gates, there are other `basic' building blocks for constructing SI circuits. These elements are `basic' in the sense that they are normally not decomposed into smaller units in logic designs, although their implementation may be based on smaller units such as basic logic gates or transistors. These elements are assumed to satisfy some properties such as speed independence or delay insensitivity by themselves. The followings gave a few of them and their DILL specifications.

Wires are most simplest components. They are not needed for SI circuits as delay on wires are assumed to be zero. But when DI or QDI circuits are transformed to SI, some of the wires should be explicitly specified to introduce delays. Suppose the input of a wire is *A* and output is *B*, When a wire is not stable, i.e. there is a pending output, the input has to wait until the output changes, otherwise the component will not be speed independent.

```
process Wire [A, B] (dtA : Bit) : noexit :=
  A ? newA : Bit [dtA ne newA];                (* accept input *)
  B ! newA;                                    (* output *)
  Wire [A, B] (newA)                           (* continue *)
endproc (* Wire *)
```

For the rest of the components, a shorthand notation is exploited to save space. In the notation a process definition is prefixed with ':=', and every LOTOS gate can represent either positive or negative signal transitions. For example, the *Wire* component now looks like:

Wire [A, B] := A; B; Wire [A, B]

Fork components are also necessary when a DI or QDI circuit is transformed to SI. A fork has one input *Ip* and two output *Op1, Op2*. The value on input *Ip* is fanned out to *Op1 and Op2*. Because of the delay on wires,the two outputs may occur at different times. New input has to wait until both outputs have been produced.

Fork [Ip, Op1, Op2] := Ip; (Op1; **exit** ||| Op2; **exit**) ≫ Fork [Ip, Op1, Op2]

C-Elements are very important elements in asynchronous design. A C-Element serves as a transition synchroniser in asynchronous design because the output can only change after both inputs have changed. For this reason, it is sometime also called *Join Element*. Precisely, a C-Element has two inputs *A, B* and an output *C*. *C* changes to *1* when both inputs have changed to *1*, and changes to *0* when both of them have changed to *0*.

C-Element [A, B, C] := (A; **exit** ||| B; **exit**) ≫ (C; C-Element [A, B, C])

Merge components `merges' signals on the input ports to the output. Each merge component has two inputs *Ip1, Ip2* and one output *Op*.

Merge [Ip1, Ip2, Op] :=
Ip1; Op; Merge [Ip1, Ip2, Op] [] Ip2; Op; Merge [Ip1, Ip2, Op]

Selectors nondeterministically produce output on either *Op1* or *Op2* after receive an input. A selector has one input *Ip* and two outputs *Ip1, Ip2*.

Selector [Ip, Op1, Op2] :=
Ip ; (**i**; Op1; **exit** [] **i**; Op2; **exit**)≫ Selector [Ip, Op1, Op2]

Sequencers have three inputs *Ip1, Ip2, N* and two outputs *Op1, Op2*. They wait for a signal on at least one of the Ipi (i=1, 2) inputs. Having received input signals on Ipi (i=1, 2) and N, the sequencer produces a signal on output Opi (i=1, 2).

Sequencer [Ip1, Ip2, N, Op1, Op2] :=
(S1 [Ip1, Op1] ||| S2 [Ip2, Op2]) |[Op1, Op2]| S3 [N, Op1, Op2]
where
S1 [Ip1, Op1] := Ip1; Op1; S1 [Ip1, Op1]
S2 [Ip2, Op2] := Ip2; Op2; S2 [Ip2, Op2]
S3 [N, Op1, Op2] := N; (**i**; Op1; S3[N, Op1, Op2] [] **i**; Op2; S3 [N, Op1, Op2])

Latches are the storage elements in asynchronous circuits. A latch has three inputs *Ip1, Ip12*, *C*, and two outputs *Op1* and *Op2*. It waits for a signal on exactly one of the Ipi (i=1, 2) inputs and a signal on the C input. In contrast to a Sequencer, the environment must guarantee mutual exclusion of the inputs Ipi (i=1, 2). Having received input signals on Ipi (i=1, 2) and *C*, a latch produces a signal on output Opi (i=1, 2).

Latch[Ip1, Ip2, C, Op1, Op2] :=
((Ip1; **exit** ||| C; **exit**) ≫ Op1; Latch [Ip1, Ip2, C, Op1, Op2])
[]
((Ip2; **exit** ||| C; **exit**) ≫ Op2; Latch [Ip1, Ip2, C, Op1, Op2])

RGD Arbiters have four inputs *r1, d1, r2*, and *d2* and two outputs *g1* and *g2*. For each i in 1,2, signal starts with *ri*, followed by an acknowledgment of *gi*, then concurrently *di* and *ri*. The intervals from *g1* to *d1* and from *g2* to it d2 are mutually exclusive. RGD stands for *Request* (ri), *Grant* (gi), and *Done* (di).

When a RGD Arbiter receives two requests, it will grant exactly one of them (and delay the other). The specification leaves the choice open.

RGD [R1,G1, D1, R2, G2, D2] :=
(S1 [R1, G1] ||| S2 [R2, G2]) |[G1, G2]| S3 [G1, D1, G2, D2]
where
S1 [R1, G1] := R1; G1; S1 [R1, G1]
S2 [R2, G2] := R2; G2; S2 [R2, G2]
S3 [G1, D1, G2, D2] := (**i**; G1; D1; S3[G1, D1, G2, D2])
[]
(**i**; G2; D2; S3[G1, D1, G2, D2])

3.4.5 Input Receptiveness

For convenience, the thesis has so far used the terms *input events* and *output events*. The more accurate phrases however should be *events corresponding to input ports (or output ports).* Since LOTOS never makes a difference between input and output events in its semantics, all events are treated equally. In LOTOS, communication between processes is based on symmetric synchronisation at a gate. Thus an event can happen only when all processes offer events at this gate. If, however, one of the processes is not able to do so, other processes just wait there, or participate in other events if possible. In the second case, the event does not occur.

As is well known, digital hardware makes a clear difference between inputs and outputs. Signals come to inputs and are produced on outputs. A component can never refuse input signals, and output signals it produces can never be blocked by others.

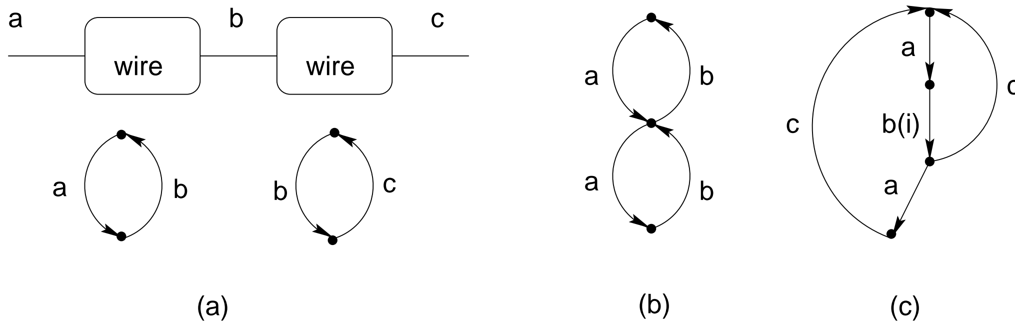


Figure 12: Two wires in series

If LOTOS events model physical signal transitions, as has been done for specifying asynchronous circuits, a DILL specification and the real circuit may have different behaviour when inputs are allowed to be refused. In section 3.4.3, behaviour of basic logic gates was defined for only desirable input situations. It is assumed that undesirable inputs are not allowed in order to respect the requirement of semi-modularity. This follows the convention of writing LOTOS specifications that only desirable behaviour is specified. There is no need to specify undesirable behaviour because it cannot happen even though proposed by environment. When this convention is used in the context of digital circuits, an input transition which will definitely happen in real world but will not happen in a DILL specification. Consequently, the behaviour of a DILL specification is just a subset of real behaviour, so analysis based on the specification is not exact. Especially if no problem is found in a DILL model, it does not necessarily mean that there is no problem in the real circuit.

To be more concrete, think about a very simple circuit which just has two wires connected in series as shown in figure 12(a). The specification of wires is according to section 3.4.4, which is a partial specification. The circuit is not speed independent under the environment shown in figure 12(b), because if a second *a* comes before the first *wire* produces its output *b*, the behaviour of the circuit is undefined. However from experience there is no way to highlight this speed dependency since the circuit behaviour (figure 12(c)) is observationally equivalent to figure 12(b). The reason is that the DILL specification can refuse the second input *a* when the first *wire* is not ready to accept it, while the real circuit cannot.

When a specification is input receptive, in other words when every input is allowed in all states, the DILL model can represent the real circuit faithfully. However input receptive specification is not available for most of the basic building blocks of SI circuits as behaviour with unexpected inputs is unknown. One way to shorten the gap is by explicitly introducing **stop** behaviour when unexpected inputs happen. Here unexpected inputs are regarded as `evil' and their appearance means something is wrong in the circuits, which is indicated by deadlock. Another possible way is to treat an unexpected input as `benign'. For example, when it happens a circuit does nothing but just stays in the same state. For SI circuits, the former solution seems better considering that unexpected inputs are usually undesirable ones. Thus this solution is adopted in DILL when partial specification cannot meet validation requirements.

Finally, it should be pointed out that input receptiveness is not so important in synchronous circuits because in most cases, the environment can guarantee there is no unexpected input.

3.4.6 Input Quasi-Receptiveness

Analysing SI circuits based on partial specifications of building blocks has the disadvantage of not being exact. If more accuracy of analysis is sought, *input quasi-receptive* specification of these blocks should be used.

Informally, a DILL specification is *input quasi-receptive* if it can always participate in all input events, except when it is in a deadlock state. Before a formal definition of input quasi-receptiveness is given, consider the simple example of the *wire* component.

Apparently, specification of *wire* in section 3.4.4 is partial in that input *A* is not allowed when the *wire* wants to produce its output. An input quasi-receptive specification can thus be obtained by adding a choice

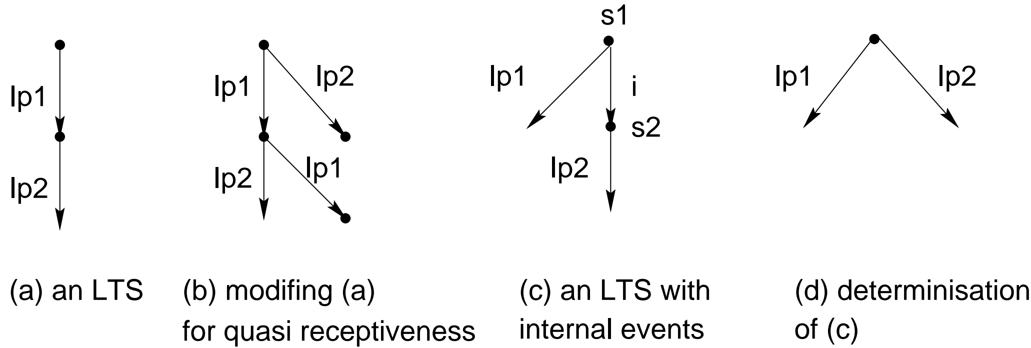


Figure 13: LTS with internal events

option when there is a potential output. The choice option is made up of forbidden input events followed by a deadlock state.

```
process Wire [A, B] (dtA : Bit) : noexit :=
  A ? newA : Bit [dtA ne newA];                        (* new input *)
  (B ! newA; exit (newA)                               * new output *)
  []                                                   (* or *)
    A ? newnewA : bit [newnewA ne newA];               (* forbidden input *)
    stop)                                              (* deadlock *)
  >>
  accept newA : Bit in
  Wire [A, B] (newA)                                   (* continue *)
endproc (* Wire *)
```

However, for those components which can not be simply specified using sequence (;) and choice ([]) operators, such as *Sequencer* or *Latch* in section 3.4.4, writing an input quasi-receptive specification is not straightforward. In this case, a partial specification is used to generate the corresponding LTS (Labelled Transition System). An LTS is actually a LOTOS specification in form of sequence and choice operators, therefore input quasi-receptive specification can be obtained by modifying the LTS. Precisely, for each state which cannot participate in all input events, outgoing edges are added which are labelled with these missed inputs and which lead to deadlock state. An example is shown in figure 13(a) and (b). Note that in the figure the input set is *Ip1, Ip2*.

This method works very well for LTSs without internal events *i*. But for those containing *i* events, things become very subtle. For example in figure 13(c), state *s1* cannot engage in event *Ip2*, but one cannot simply think that *Ip2* is rejected in this state, as the component may decide to accept it at state *s2* through an internal event. Such kinds of peculiar situation may not be the intention of specifiers, but can appear in an LTS through hide operations.

Notice that when a specification is understood from the point of view of input receptiveness, internal events seems to lose their necessity. Take the example of the RGD arbiter. Internal events are used to indicate that the component will decide which outputs (*g1* or *g2*) should be produced, and that its environment has no influence on this decision. If the internal events are omitted, the environment can choose which output to accept and which one to refuse. However, if the environment is input receptive, which means it can always receive all inputs (the outputs of the RGD), it loses its selective power. Whichever the output produced by the RGD, the environment just accepts it. In other words, no matter if the internal events are specified or not, the environment cannot affect the decision made by the RGD.

Based on this observation, LTSs with internal events are determinised before outgoing edges are added (figure 13(d)) to obtain input quasi-receptive specifications

Definition 3.1 (Deterministic LTS) *An LTS* $p = < S, L, T, s0 >$ *is deterministic if* $\forall\ s \in S,\ s \stackrel{i}{\not\rightarrow}$ *and* $\forall\ a \in L$, *s after a contains at most 1 element.*

Definition 3.2 (Determinization of LTS) *Let* $p =< S, L, T, s0 >$ *be an LTS.* $\mathcal{P}(S)$ *is the powerset of S. The deterministic LTS* $p_d =< S_d, L, T_d, s0_d >$ *can be obtained by:*

$$
\begin{array}{lll}
S_d & =_{def} & \mathcal{P}(S)\backslash\{\emptyset\} \\
T_d & =_{def} & \{q \xrightarrow{a} q' \mid a \in L, q, q' \in S_d, q' = \{s' \in S \mid \exists s \in q : s \stackrel{a}{\Longrightarrow} s'\}\} \\
s0_d & =_{def} & \{s' \in S \mid s_0 \stackrel{\epsilon}{\Longrightarrow} s'\}
\end{array}
$$

Now, the formal definition of input quasi-receptive can be given.

Definition 3.3 (Input Quasi-Receptive) *Let L be partitioned into* L_I *and* L_U*, and let* $c =< S, L, T, s0 >$ *be a deterministic labelled transition system. c is input quasi-receptive if for every state* $s \in S$*, either for all* $a \in L_I \cup L_U$*,* $s \stackrel{a}{\not\Rightarrow}$*, or* $L_I \subseteq \{a \mid s \stackrel{a}{\Rightarrow}\}$*. If c is not a deterministic LTS, it should be determinised according to the previous definition.*

From the above definition, an LTS is input quasi-receptive if after determinization, all its states, except the terminal ones, can engaged in all the events in L_I.

3.4.7 Case Study: Specifying a FIFO

In this section, an asynchronous FIFO (First In First Out) is specified. The FIFO has two inputs *InT, InF* and two outputs *OutT, OutF*. Its input and output data conform to *dual rail* encoding in which representing one bit needs two signal lines. When *InT* (*OutT*) is *1* and *InF* (*OutF*) is *0*, the transmitted (received) data is *1*. Similarly when *InT* (*OutT*) is *0* and *InF* (*OutF*) is *1*, the transmitted (received) data is *0*. When the signal on both lines is *0*, it indicates *idle*, which means no valid data on the lines. Lines have to be reset to *idle* between two transmissions.

Suppose a FIFO with one stage (figure 14(a)) is initially empty. It can accept either *1* or *0* by raising *InT* or *InF*. The accepted data can be delivered to its environment by output ports. After one successful transmission, the raised input and output ports return to *0* to wait for other data. The behaviour of one stage can be easily specified:

```
process Stage [InT, InF, OutT, OutF] :noexit :=
 InT ! 1 of bit; OutT ! 1 of bit;                    (* transmit 1 *)
 InT ! 0 of bit; OutT ! 0 of bit;                    (* go to idle *)
 Stage [InT, InF, OutT, OutF]                        (* continue *)
[]
 InF ! 1 of bit; OutF ! 1 of bit;                    (* transmit 0 *)
 InF ! 0 of bit; OutF ! 0 of bit;                    (* go to idle *)
 Stage [InT, InF, OutT, OutF]                        (* continue *)
endproc
```

The behaviour of a FIFO of more than one stage can be obtained by composing several stages. For simplicity, a FIFO with two stages (figure 14(b)) is specified with:

```
process SpecFIFO [InT, InF, OutT, OutF]
 hide i1, i2 in
  Stage [InT, Inf, i1, i2]
 |[i1, i2]|
  Stage [i1, i2, OutT, OutF]
endproc
```

A possible implementation of one stage is given in figure 15. Apart from the data path, there are another two lines controlling the data transmission. *Req* comes from the environment of a stage; it indicates that environment has valid data to transfer. The *Ack* line goes to the environment, indicating that the stage is empty and is thus ready to receive new data. Both of these control signal are high active. The implementation use two *C-Elements* and a *Nor2* gate. Initially both *Req* and *Ack* are *1*. When there is valid data on *InT* or *InF*, it is passed to *OutT* or *OutF*. At the same time, *Req* should be reset to *0* until *InT* or *InF* returns to the idle state. After receiving data on *OutT* or *OutF*, the *Ack* reset to *0* indicates that the stage is full. When the data on output lines is fetched, output returns to the idle state and is ready for the next transmission. The corresponding DILL specification of this cell is as follows:

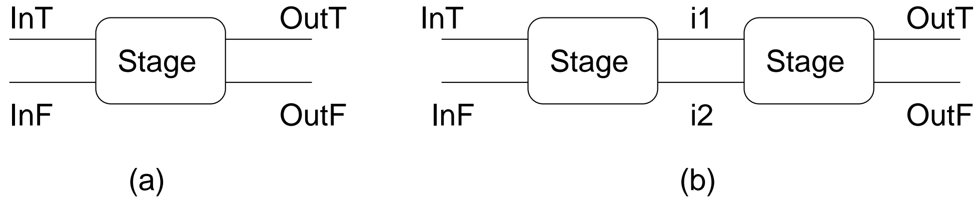


Figure 14: FIFO with one stage and with two stages

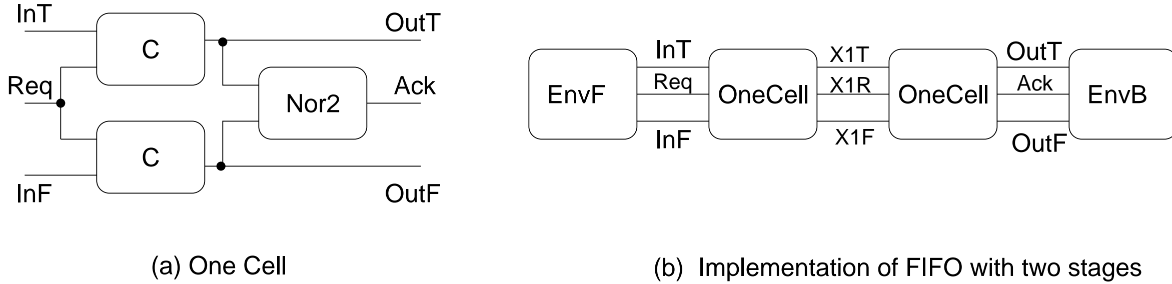


Figure 15: Implementation of a cell of FIFO

process OneCell [InT, InF, OutT, OutF, Req, Ack] : **noexit** :=
(CElement [InT, Req, OutT] (0 **of** Bit, 1 **of** Bit, 0 **of** Bit)
|[Req]|
CElement [InF, Req, OuTF] (0 **of** Bit, 1 **of** Bit, 0 **of** Bit)
)
|[OutT, OutF]|
Nor2 [OutT, OutF, Ack] (0 **of** Bit, 0 **of** Bit, 1 **of** Bit)
endproc

To ensure a FIFO works correctly, the environment has to be coordinated. For example, it should provide correct input data according to the dual rail encoding. To make things easier, it is convenient to think about the environment in two parts: EnvF and EnvB. EnvF is a data provider which is always ready to produce data. EnvB is a data consumer which can always accept data.

process EnvF [Req, InT, InF] : **noexit** :=
InT ! 1 **of** bit; Req ! 0 **of** bit; InT ! 0 **of** bit; Req ! 1 **of** bit; (* provide 1 *)
EnvF [Req, InT, InF] (* continue *)
[]
InF ! 1 **of** bit; Req ! 0 **of** bit; InF ! 0 **of** bit; Req ! 1 **of** bit; (* provide 0 *)
EnvF [Req, InT, InF] (* continue *)
endproc

process EnvB [Ack, OutT, OutF] : **noexit** :=
OutT ! 1 **of** bit; Ack ! 0 **of** bit; OutT ! 0 **of** bit; Ack ! 1 **of** bit; (* accept 1 *)
EnvB [Ack, OutT, OutF] (* continue *)
[]
OutF ! 1 **of** bit; Ack ! 0 **of** bit; OutF ! 0 **of** bit; Ack ! 1 **of** bit; (* accept 0 *)
EnvB [Ack, OutT, OutF] (* continue *)

A two-stage FIFO can then be implemented:

process TwoStages [InT, InF, OutT, OutF] : **noexit** :=
hide Req, X1T, X1F, X1R, Ack **in**
EnvF [Req, InT, InF]
|[Req, InT, InF]|

```
  OneCell [InT, InF, X1T, X1F, X1R, Req]
|[X1T, X1F, X1R]|
  OneCell [X1T, X1F, OutT, OutF, Ack, X1R]
|[Ack, OutT, OutF]|
  EnvB [Ack, OutT, OutF]
endproc
```

In the next chapter, the implementation will be verified against its specification. When speed independence needs to be verified, each building block, including the environment, should be specified in the input quasi-receptive style. The DILL library already contains such specifications of basic building blocks. *EnvB_QR* is the input quasi-receptive specification of *EnvB*. *EnvF* has no inputs so there is no need to modify it.

```
process EnvB_QR [Ack, OutT, OutF] : noexit :=
  OutT ! 1; (Ack ! 0 ; (OutT ! 0; (Ack ! 1; EnvB_QR [Ack, OutT, OutF]
                                    []OutT ! 1; stop
                                    []OutF ! 1; stop)
                        []Ack ! 1; stop
                        []OutF ! 0 ; stop)
             []OutT ! 0; stop
             []OutF ! 0; stop)
[]
  OutF ! 1; (Ack ! 0; (OutF ! 0; (Ack ! 1; EnvB_QR [Ack, OutT, OutF]
                                   []OutT ! 1; stop
                                   []OutF ! 1; stop)
                       []Ack ! 1; stop
                       []OutT !0; stop)
             []OutT ! 0; stop
             []OutF ! 0; stop)
[]
  Ack ! 0; stop
endproc (* EnvB_QR *)
```

3.5 Related Work

Hardware Description Languages (HDLs) were initially designed to cope with the inefficiency of circuit diagrams when the size of circuits became more and more large. They were subsequently used in simulation, synthesis and verification of digital logics. The most popular HDLs used in industry are perhaps VHDL, Verilog and ELLA. These languages are very expressive and can give very detailed models of real hardware circuits. But circuits described in these languages cannot be formally analysed because there is no formal semantics associated with them.

Some formal languages are specifically designed for specifying circuits, such as Ruby [JS90], CIRCAL, Synchronous Transitions [Sta97], DI-algebra [JU93] and so on. Many other general purpose formal languages or notations are also applied in the area of hardware specifications. To name a few, these includes HOL [HG92], CSP [Hoa85], Occam [TTW97], and trace theory [Dil89]. Some of them can just deal with synchronous circuits, such as Ruby and HOL. Others are mainly employed to tackle asynchronous circuits, such as DI-algebra and CSP. CIRCAL and Synchronous Transition have been used in both areas. Among these formalisms, DILL most closely resembles CIRCAL in that both have a behavioural basis in process algebra, and both have been used in synchronous and asynchronous circuit designs. In fact, DILL was inspired by the success of CIRCAL. However, the integrated data types in LOTOS makes it much more expressive than CIRCAL. In the authors' experience, DILL can be used successfully at a variety of abstraction levels. However, CIRCAL appears to be less effective at higher levels. For example, describing the behaviour of a synchronous circuit in CIRCAL requires the corresponding Mealy or Moore machines to be defined manually, and then translated into the CIRCAL notations. This makes CIRCAL almost impossible to specify relatively complicated behaviour.

3.6 Conclusion

This chapter provides the LOTOS models of synchronous and asynchronous circuits. The models are the basis of further analysis of digital circuits.

Specification of basic logic gates represents the basic modelling technique for DILL. In this chapter there were four different models developed for basic logic gates. Initially a unified model was preferred because it can be used in both synchronous and asynchronous circuits. This idea resulted in the specification in section 3.2. However, later investigation revealed that the same components may need different models in different environments. One of the such examples are the models in 3.2 and 3.4.3, with the latter being specially developed for validating semi-modularity of asynchronous circuits. As will be seen in chapter 6, a timed model of basic logic gates will also be provided for analysing behaviour related to quantitative timing.

Process algebra, such as CCS, CSP, Circal, and LOTOS have been used in specifying and analysing digital circuits for many years. However, the thesis is the first one which clearly points out the gap between the behaviour modelled by process algebras and the behaviour of real circuits. Moreover, it reveals that when an event models a signal transition, the behaviour of a LOTOS specification is just a subset of the behaviour of a real circuit, resulting the analysis based on LOTOS models being inaccurate . The author of [Gop92], who used CCS to model asynchronous circuits, also realised the gap, but the solution proposed is not complete: to get a specification which is similar to the input quasi-receptive style employed here, all components are still specified in normal style but their inputs are preceded with an artificial wire which is input quasi-receptive. This solution cannot deal with all the unexpected input situations and thus has limited usage. In [CT97], CCS also used to specify and analyse asynchronous circuits. The authors suggested to use the 'quenching' specifications to bridge the gap. This solution is actually identical to the first model of basic logic gates discussed in section 3.2, where the pending outputs can be pre-empted by new inputs. Their solution is therefore covered by this thesis.

In this thesis, in order to discover violations of speed-independence (or really semi-modularity), components are specified in input quasi-receptive style or if possible, input receptive style. Except for explicitly using **stop**, there is an alternative solution which is similar to the one used in chapter 6, that is using an extra gate *Err* to indicate violation. This unfortunately would result in much bigger state space comparing with the solution adopted here, simply because every component should have an extra *Err* gate and these gates interleave with each other. In chapter 6, analysis is mainly achieved by simulation and testing, state space is not a severe problem in these two validation methods.

4 Verification of Digital Logic Circuits

This chapter applies formal verification techniques to verifying DILL specifications of digital circuits. In section 4.1, background knowledge about formal verification of LOTOS specificatinos was introduced, with the focus on the temporal logic ACTL (Action based CTL [DV90]) and relations between LTSs. ACTL will be used to express properties of circuits, and relations will be used to compare specifications at different abstraction levels. The following sections, section 4.2 and section 4.3, present how the DILL specifications of synchronous and asynchronous circuits are verified respectively. Verifying synchronous circuits is straightforward, thanks to the DILL model developed in section 3.3. Conventional verification techniques, such as model checking and equivalence checking, can be conducted using general LOTOS tools. Verifying asynchronous circuits needs more consideration, such as the input receptiveness of components and the importance of the environment of asynchronous circuits. These extra considerations are necessary mainly because input and output signals are different in real circuits but are treated equally in the LOTOS model. In section 4.3.3, new relations between LTSs are defined by taking the difference between inputs and outputs into account. Throughout the presentation of the chapter, examples and case studies are provided to illustrate the approaches.

4.1 Background

This section gives the preliminary knowledge required to verify DILL specifications.

4.1.1 What is to be verified

Formal verification comprises techniques used to prove the correctness of the models of a real-world system. According to [Sta93] there are mainly three different verification tasks when talking about verification of circuits:

1. Verifying that a circuit specification is what it should be, termed *requirements capture*
2. Verifying that a given implementation behaves identically to a given specification, termed *implementation verification*
3. Verifying important properties of a given implementation, termed *design verification*

In the phase of *requirements capture*, higher level specifications of circuits are analysed to see if they satisfy some requirements. Requirements capture can be performed either formally or informally, depending on how the requirements are expressed. In figure 1, it is also termed specification validation. *Implementation verification* involves comparing two specifications of the same circuit. By convention, the higher level specification is termed the *specification* of a circuit, while the lower level one is the *implementation* of the circuit. A relation has to be defined in order to compare the implementation with the specification. Although task 2 emphasises the identity, weaker relations are also used in practice. Like implementation verification, *design verification* also aims to verify the correctness of lower level implementations of circuits, but it focuses on the properties of implementations rather than their relationships with higher level specifications.

The properties that different circuits possess may vary. For convenience these properties are divided into different categories, such as freedom from deadlock, freedom from livelock, safety and liveness. Informally:

- deadlock means that a system can evolve into a state from which no further action is possible.
- livelock means that a system may get into an internal loop and make no further progress in terms of visible inputs and outputs.
- a safety property means that nothing bad will happen during the progress of a system.
- a liveness property means that something good will eventually happen.

To formally express these properties, formal specifications are required. Many hardware verification systems formulate properties in a temporal logic or modal logic such as CTL (Computational Tree Logic [CES86]), ACTL, HML (Hennessy-Milner Logic [HM80]) or μ-calculus [Lar90]. Other formalisms are also available. For example, to verify LOTOS specifications, LTSs are sometimes used to express properties.

The method of formally verifying whether a finite-state model satisfies some properties is termed *model checking*. Efficient algorithms [BCM+92] have been developed for model checking temporal logic formulae.

For implementation verification, a relation should be formally defined to indicate that in what sense an implementation is correct with respect to a specification. LOTOS inherits abundant relations from process algebra. Most of them belong to three categories: preorder, equivalence and congruence. An equivalence holds when two specifications have exactly the same behaviour. A congruence relation requires that not only are two specifications equivalent, but also one can substitute for the other in all circumstances. Based on the way behaviour is observed, each category contains a spectrum of relations. Choosing a suitable relation for verification is sometimes not easy. It needs reasonable knowledge of both specification and implementation, and also depends on the intention of verification. As a rule of thumb, if a specification is non-deterministic, then a preorder relation should be used when verifying one of its deterministic implementations. If an implementation is a refinement of a specification, e.g. by giving more detail about how to build the system, an equivalence relation might be preferred. If the implementation is to be used within a larger system, then a congruence relation has to be considered.

Properties and relations are also related. Some relations respect certain properties while the others do not. For example, trace equivalence does not preserve deadlock freedom, while observational equivalence does. Observational equivalence, on the other hand, does not preserve livelock freedom. This factor should also be taken into account when conducting verification.

The existing LOTOS verification techniques and tools support all the three verification tasks mentioned at the beginning of this section. For example, temporal logic model checking can be employed to fulfil task 1 and task 3. Verification of equivalence or preorder relations, which has been intensively studied for Labelled Transition Systems, can be used to solve task 2. When a property is expressed in the form of an LTS, relation checking can also be used for task 1 and task 3. The following two sections introduce the temporal logic ACTL and several relations used for verifying DILL specifications.

4.1.2 Temporal Logic and ACTL

In the preceding section, it was mentioned that modal and temporal logics are used to specify properties of circuits. Modal logic is an extension of propositional calculus. In addition to the usual propositional operators $\wedge, \vee, \neg$, etc. there are also modalities which express the `modes' of truth, such as *necessarily true* or *possibly true*. Temporal logics extend modal logics with timing operators, which indicates *when* a statement is true. Four temporal operators are commonly used in various temporal logics: X (is true at the next time instant), F (is eventually true), G (is always true from now on) and U (is true until $\cdots$).

Traditionally the properties of systems modelled in process algebra are expressed in the modal logic HML and μ-calculus. Both logics are interpreted over labelled transition systems (LTS), which is also the semantic model of process algebra.

However, because the expressive power of HML is limited, and μ-calculus requires exponential time for model checking, this thesis employs the temporal logic ACTL to specify properties. It is shown in [DV90] that ACTL is more expressive than HML, and the time complexity of ACTL model checking is linear in both the length of formulae and the size of the models to be analysed.

Most temporal logics developed so far are state-based. These logics are interpreted over a *Kripke Structure*. The structure is essentially a finite state-transition system of which each state is labelled with a set of atomic propositions. All formulae refer to the states in Kripke Structure. Among such logics, the most popular one is perhaps CTL, which has linear time for model checking and is adopted in many well-known hardware verification tools such as SMV [McM93], VIS [BH+96], etc. But these state-based temporal logics cannot be used to express the properties of LTSs, because in LTSs only transitions are labelled and there is no proposition associated with states. ACTL was developed in [DV90] to overcome the problem. The rest of this section describes it.

Definition 4.1 (Path)
Let $L \in \mathcal{LTS}$ <S, A, T, s0 >, a sequence (s0, a0, s1), (s1, a1, s2), ... is called a path from s0 if there is $s0 \xrightarrow{a0} s1 \xrightarrow{a1} s2 \ldots$, *of which* $a_i (i = 0, 1, \ldots) \in A \cup \tau$.

ACTL is analogous to CTL but interpreted over actions. In order to express the predicates over actions, a small auxiliary logic of actions is needed. If *A* is the set of actions, then the action formulae $a \in A \cup \{\tau\}$ [3] are defined by:

$$\alpha ::= \text{true} \mid a \mid \neg\alpha \mid \alpha \wedge \alpha$$

The usual derived boolean operators are also allowed: there are *true* for $\neg(a \wedge \neg a)$, *false* for $\neg$true, $a \vee a$ for $\neg(\neg a \wedge \neg a)$ and so on.

Action formulae α are interpreted over the actions *a* ($a \in A \cup \{\tau\}$) of an LTS model *M* = <*S, A, T, s0*>. The *satisfaction* of an action formula α by an action *a* ($a \in A \cup \{\tau\}$), denoted by $a \models_M \alpha$ (or $a \models \alpha$ when model *M* is understood), is defined inductively by:

$$\begin{array}{lll} a \models \text{true} & \text{always;} & \\ a \models b & \text{iff} & a = b; \\ a \models \neg\alpha & \text{iff} & a \not\models \alpha; \\ a \models \alpha \wedge \alpha' & \text{iff} & a \models \alpha \text{ and } a \models \alpha'. \end{array}$$

The syntax of an ACTL formula φ is defined by the following grammar:

$$\begin{array}{lll} \varphi & ::= & \text{true} \mid \neg\varphi \mid \varphi \wedge \varphi \mid EX_\alpha\varphi \mid AX_\alpha\varphi \\ & & \mid E[\varphi_\alpha U \varphi] \mid E[\varphi_\alpha U_\alpha \varphi] \mid A[\varphi_\alpha U \varphi] \mid A[\varphi_\alpha U_\alpha \varphi] \end{array}$$

The satisfaction of an ACTL formula φ by a state $s \in S$ of an LTS *M* = <*S, A, T, s0*>, written $s \models_M \varphi$ (or $s \models \varphi$ when *M* is understood), can be defined inductively:

$$\begin{array}{lll} s \models \text{true} & \text{always;} & \\ s \models \neg\varphi & \text{iff} & s \not\models \varphi; \\ s \models \varphi \wedge \varphi' & \text{iff} & s \models \varphi \text{ and } s \models \varphi'; \\ s \models EX_\alpha\varphi & \text{iff} & \exists s \xrightarrow{a} s' \in T \text{ such that } a \models \alpha \text{ and } s' \models \varphi; \\ s \models AX_\alpha\varphi & \text{iff} & \forall s \xrightarrow{a} s' \in T, a \models \alpha \text{ and } s' \models \varphi; \\ s \models E[\varphi_\alpha U \varphi'] & \text{iff} & \exists s (= s0) \xrightarrow{a0} s1 \xrightarrow{a1} \ldots \in \text{path}(s) \\ & & \exists k \geq 0 \text{ such that } s_k \models \varphi' \text{ and } \forall i \in [0; k-1], s_i \models \varphi \text{ and } a_i \models \alpha \\ s \models E[\varphi_\alpha U_{\alpha'} \varphi'] & \text{iff} & \exists s (= s0) \xrightarrow{a0} s1 \xrightarrow{a1} \ldots \in \text{path}(s) \\ & & \exists k > 0 \text{ such that } s_k \models \varphi' \text{ and } \forall i \in [0; k-1], s_i \models \varphi \text{ and} \\ & & \forall j \in [0; k-2], a_j \models \alpha \text{ and } a_{k-1} \models \alpha'; \\ s \models A[\varphi_\alpha U \varphi'] & \text{iff} & \forall s (= s0) \xrightarrow{a0} s1 \xrightarrow{a1} \ldots \in \text{path}(s) \\ & & \exists k \geq 0 \text{ such that } s_k \models \varphi' \text{ and } \forall i \in [0; k-1], s_i \models \varphi \text{ and } a_i \models \alpha \\ s \models A[\varphi_\alpha U_{\alpha'} \varphi'] & \text{iff} & \forall s (= s0) \xrightarrow{a0} s1 \xrightarrow{a1} \ldots \in \text{path}(s) \\ & & \exists k > 0 \text{ such that } s_k \models \varphi' \text{ and } \forall i \in [0; k-1], s_i \models \varphi \text{ and} \\ & & \forall j \in [0; k-2], a_j \models \alpha \text{ and } a_{k-1} \models \alpha'; \end{array}$$

Besides the usual derived boolean operators, the following are some useful modalities:

$$\begin{array}{lll} \langle\alpha\rangle\varphi & = & EX_\alpha\varphi \\ [\alpha]\varphi & = & \neg\langle\alpha\rangle\neg\varphi \\ EF_\alpha\varphi & = & E[\text{true}_\alpha U \varphi] \\ AF_\alpha\varphi & = & A[\text{true}_\alpha U \varphi] \\ EG_\alpha\varphi & = & \neg AF_\alpha\neg\varphi \\ AG_\alpha\varphi & = & \neg EF_\alpha\neg\varphi \end{array}$$

Like in CTL, all the legal ACTL formulae are state formulae, which are true if the current state satisfies the formula or false if otherwise. In the ACTL formulae, *A* and *E* are *path quantifiers* which define whether a property of current state should be true for all its possible paths (*A*) or only for some path (*E*). The basic temporal operators of ACTL are *X* and *U*; *F* and *G* are derived operators. The following gives the intuitive meaning of several common ACTL formulae:

[3] The original ACTL does not include τ in the action set.

- $EX_\alpha\varphi$ (i.e. $\langle\alpha\rangle\varphi$) is true of a state if it has an action satisfying α and the action leads to a state satisfying φ. This is also the *diamond* operator in HML.
- $AX_\alpha\varphi$ is true of a state if the state can only do actions satisfying α and the actions lead to states satisfying φ.
- $[a]\varphi$ is true of a state if all its actions which satisfy α lead to states satisfying φ. This is also the *Box* operator in HML.
- $EF_\alpha\varphi$ is true of a state if some of its paths can begin with a series of actions satisfying α then reach a state satisfying φ. If the state itself satisfies φ, then the formula is also true.
- $AF_\alpha\varphi$ is true of a state if all its paths can begin with a series of actions satisfying α then reach a state satisfying φ. If the state itself satisfies φ, then the formula is also true.
- $EG\varphi$ is true of a state if there exists one or more paths on which all states satisfy φ.
- $AG\varphi$ is true of a state if all the states on all its paths satisfy φ. It is also said that φ is satisfied globally.

4.1.3 Relations between LTSs

The operational semantics of LOTOS is defined based on labelled transition systems. In this section, several common relations between LTSs are presented. More relations can be found in [Gla90, Gla93] and [Nic87], where a spectrum of equivalent and congruent relations are compared in terms of distinguishing power.

Definition 4.2 (Strong Equivalence)
A relation, $R \subseteq \mathcal{LTS}(\mathcal{L}) \times \mathcal{LTS}(\mathcal{L})$ is a strong bisimulation if $(P, Q) \in R$ implies, $\forall a \in L \cup \tau$, that:

- *if $\exists P' : P \xrightarrow{a} P'$ then $\exists Q' : Q \xrightarrow{a} Q'$ with $(P', Q') \in R$, and*
- *if $\exists Q' : Q \xrightarrow{a} Q'$ then $\exists P' : P \xrightarrow{a} Q'$ with $(P', Q') \in R$.*

Two processes P, Q are strongly equivalent, written $P \sim Q$ if there exists a strong bisimulation R such that $(P, Q) \in R$. The relation $\sim$ is defined to be the largest strong bisimulation, i.e. the union of all strong bisimulations.

Strong bisimulation equivalence can distinguish more processes than any other equivalent relations. However it is usually too strong to be used in practice since it requires two processes to match each other even on internal behaviour. For example process *a; i; i; b; stop* is not strongly equivalent to process *a; i; b; stop*. Internal events are unseen to external observers, and thus are meaningless in most circumstances.

Definition 4.3 (Observational Equivalence)
A relation, $R \subseteq \mathcal{LTS}(\mathcal{L}) \times \mathcal{LTS}(\mathcal{L})$ is a weak bisimulation if $(P, Q) \in R$ implies, $\forall a \in L \cup \tau$, that:

- *if $\exists P' : P \xrightarrow{a} P'$ then $\exists Q' : Q \overset{\hat{a}}{\Rightarrow} Q'$ with $(P', Q') \in R$, and*
- *if $\exists Q' : Q \xrightarrow{a} Q'$ then $\exists P' : P \overset{\hat{a}}{\Rightarrow} Q'$ with $(P', Q') \in R$.*

Two processes P, Q are observationally equivalent, written $P \approx Q$ if there exists an weak bisimulation R such that $(P, Q) \in R$. The relation $\approx$ is defined to be the largest weak bisimulation, i.e. the union of all weak bisimulations.

In the above definition, $P \overset{\hat{a}}{\Rightarrow} Q$ has the same meaning with $P \overset{a}{\Rightarrow} Q$ when *a* is not an internal event. Otherwise it means the same as $P \overset{\epsilon}{\Rightarrow} Q$. Recall that $P \overset{\epsilon}{\Rightarrow} Q$ is defined as $P = Q$ or $P \xrightarrow{\tau \cdot \ldots \cdot \tau} Q$, and for $a \in L$, $P \overset{a}{\Rightarrow} Q$ is defined as $\exists s_1, s_2 : P \overset{\epsilon}{\Rightarrow} s_1 \xrightarrow{a} s_2 \overset{\epsilon}{\Rightarrow} Q$. From the definition, the internal events τ may be ignored when determining if two processes are observational equivalent or not. As an observer interacts

with a system through its external interface, observational equivalence is often used for characterizing systems.

Observational equivalence is not a congruence. Precisely it is not preserved by the choice operator of LOTOS. For example, although *i; b; stop* $\approx$ *b; stop*, it is not true that *a; stop [] i; b; stop* $\approx$ *a; stop [] b; stop*. The consequence is that if *i; b* is replaced by *b; stop*, the new system is not observationally equivalent to the original one.

It is known that observational equivalence is a congruence with respect to other LOTOS operators such as prefix (;), parallel ($|||$, $|\ [\cdots]\ |$) and *hide* [ISO89, Mou94]. In the DILL approach, composing components is done by putting the processes in parallel, then hiding the internal connecting ports. Observational equivalence can be used as a congruence in this circumstance. In other words, if two components *A* and *B* are proved be observationally equivalent, then a circuit which contains the *A* component can be changed to a new one by substituting *A* for *B*. The resultant circuit is still observationally equivalent to the original one.

Observational equivalence can preserve deadlock freedom. If two processes are observationally equivalent, they are both free from deadlocks or both possessing deadlocks. However, this equivalence cannot preserve livelock freedom, or liveness properties.

Definition 4.4 (Branching Bisimulation Equivalence)
A relation $R \subseteq \mathcal{LTS}(\mathcal{L}) \times \mathcal{LTS}(\mathcal{L})$ *is a branching bisimulation if* $(P,Q) \in R$ *implies,* $\forall a \in L \cup \tau$*, that*

- *if* $\exists P' : P \xrightarrow{a} P'$ *then either* $a = \tau$ *and* $(P', Q) \in R$*, or*
  $\exists$ *a path* $Q \Rightarrow Q_1 \xrightarrow{a} Q_2 \Rightarrow Q'$ *with* $(P, Q_1) \in R, (P', Q_2) \in R, (P', Q') \in R$*, and*
- *if* $\exists Q' : Q \xrightarrow{a} Q'$ *then either* $a = \tau$ *and* $(P, Q') \in R$*, or*
  $\exists$ *a path* $P \Rightarrow P_1 \xrightarrow{a} P_2 \Rightarrow P'$ *with* $(P_1, Q) \in R, (P_2, Q') \in R, (P', Q') \in R$

Two processes P, Q are branching bisimulation equivalent, written $P \approx_b Q$ *if there exists a branching bisimulation R such that* $(P,Q) \in R$*. The relation* $\approx_b$ *is defined to be the largest branching bisimulation, i.e. the union of all branching bisimulations.*

Branching bisimulation equivalence [Gla90] is stronger then observational equivalence but weaker than strong equivalence. Essentially the definition is the same as observational equivalence. The difference is that it compares the states not only at the start and finish of a τ sequence, but also the states along the τ sequences. Branching bisimulation equivalence is a congruence and preserves liveness properties when both processes are livelock free [DV90]. In other words, if two LTSs are free from livelock and are related by branching bisimulation equivalence, they satisfy exactly the same set of liveness properties.

Definition 4.5 (Simulation Preorder and Simulation Equivalence)
A relation $R \subseteq \mathcal{LTS}(\mathcal{L}) \times \mathcal{LTS}(\mathcal{L})$ *is a simulation if* $(P,Q) \in R$ *implies,* $\forall a \in L \cup \tau$*, that:*

- *if* $\exists P' : P \xrightarrow{a} P'$ *then* $\exists Q' : Q \xrightarrow{a} Q'$ *with* $(P', Q') \in R$*.*

The simulation preorder $<_s$ *is defined as the largest simulation.*
The simulation equivalence $\sim_s$ *is defined by* $\sim_s = <_s \cap (>_s)^{-1}$*.*

Definition 4.6 (Safety Preorder and Safety Equivalence)
A relation $R \subseteq \mathcal{LTS}(\mathcal{L}) \times \mathcal{LTS}(\mathcal{L})$ *is a safety simulation if* $(P,Q) \in R$ *implies for* $\forall a \in L \cup \tau$

- *if* $\exists P' : P \xrightarrow{a} P'$ *then* $\exists Q' : Q \overset{a}{\Rightarrow} Q'$ *with* $(P', Q') \in R$*.*

The safety preorder $\prec_s$ *is defined as the largest safety simulation.*
The safety equivalence $\approx_s$ *is defined by* $\approx_s = \prec_s \cap (\succ_s)^{-1}$*.*

Safety equivalence is named after its important feature: it preserves all safety property, i.e. if two LTSs are related by safety equivalence, they satisfy the exactly same set of safety properties [BFG$^+$91].

To verify hardware, testing equivalence/preorder and trace equivalence/preorder are also used. Because they are not used in the DILL approach, the formal deldfinitions are not given here. Intuitively, two LTSs are

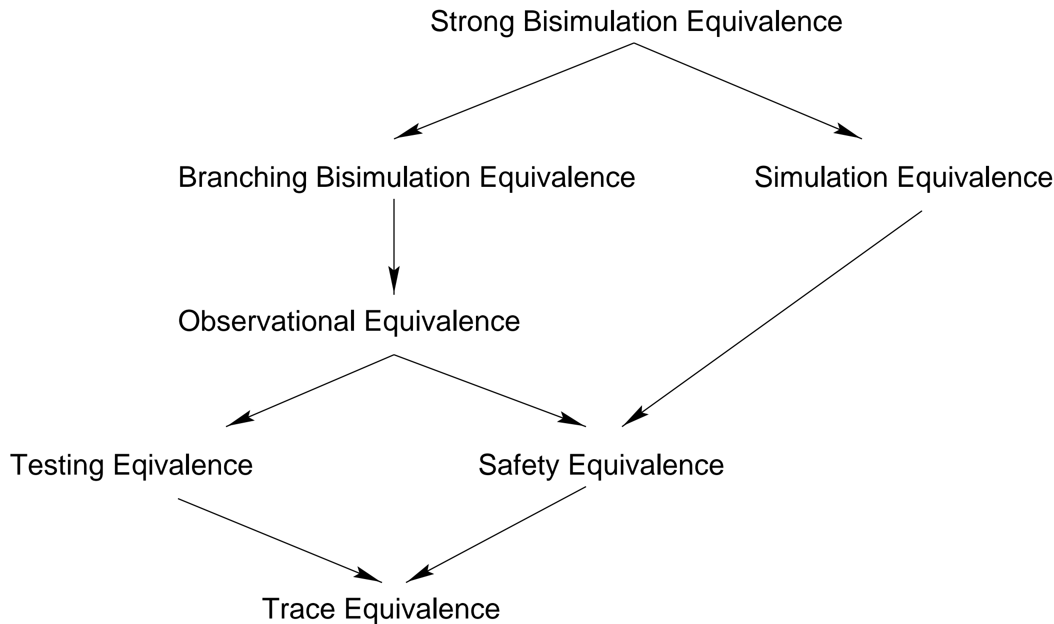


Figure 16: Different equivalence relations with different strengths

testing equivalent if any possible observer, which is also modelled as an LTS, cannot distinguish them after interacting with the two LTSs. This relation is weaker than observational equivalence. Trace equivalence is the weakest equivalent relation between LTSs; it only requires that two LTSs have the same trace set.

Figure 16 gives the relative strength of these relations. In the figure *R1* → *R2* means that *R1* is a finer relation than *R2*, i.e. *R1* is able to differentiate at least as many specifications as *R2* is. Thus according to this figure, if two relations are *strong bisimulation equivalent*, they must also be *observational equivalent*, but not vice visa.

Note that when LTSs are deterministic, all the equivalence relations in the above figure coincide.

4.1.4 CADP

CADP (Cæsar Aldébaran Development Package [FGK+96]) is an automated toolset for analysing concurrent systems expressed in LOTOS or other formalisms whose semantics are based on LTSs. It is perhaps the most comprehensive tool available to support LOTOS currently. CADP includes several tools each of which fulfil a specific functionality. Cæsar.ADT and Cæsar compile the data part and the behaviour part of a LOTOS specification respectively. The result is a finite state graph (i.e. an LTS) which describes the exhaustive behaviour of the corresponding specification. Aldébaran performs verification using the LTS or a network of LTSs (i.e. a finite state machine connecting several LTSs by LOTOS parallel and hiding operators). It is able to either compare or minimize LTSs with respect to bisimulation or simulation relations. XTL (Executable Temporal Language) is a functional-like programming language that allows the implementation of temporal logic operators. Several temporal logics such as ACTL have been embedded in XTL. The tool with the same name can be used to perform model checking XTL formulae against LTS models. To partially resolve the problem of state space explosion, CADP incorporates advanced verification techniques such as compositional generation, on-the-fly comparison, and BDD (Binary Decision Diagram) symbolic representation of LTSs. These techniques make it possible to verify relatively large specifications.

CADP also supports customized verification tools. It provides a programming interface through which the LTSs of LOTOS specifications can be manipulated. This interface is used to implement a verifier for asynchronous circuits in chapter 4 and a test generation tool in chapter 5.

4.2 Verification of Synchronous Circuits

In this section, the benchmark circuit *single pulser* will be investigated to demonstrate how synchronous circuits are verified. The behavioural and structural specifications of this circuit have been given in section 3.3.5.

As mentioned in section 3.3.5, the behavioural specification is non-deterministic since it allows two different implementations: one asserts an output pulse when the button is pressed, the other asserts the output when the button is released. The structural implementation given in the benchmark is deterministic; in particular it produces the output pulse when the button is pressed. In the sequel, properties as well as the relations between the two levels of specification will be verified by CADP.

4.2.1 Verifying Properties

First of all the basic properties, i.e. freedom from deadlock and livelock, are checked after the LTSs of both specification and implementation are generated. It is found that they both satisfy the properties.

The implementation is also required to fulfil other properties. Two of them and their corresponding ACTL formulae are listed below. In these formulae, an event with ellipsis means the corresponding signal can either be *1* or *0*. For example, *Op*... represents both *Op ! 0* and *Op ! 1*. To make the formulae more readable, $[\tau^\star\alpha]\varphi$ is used as a shorthand notation for the weak form of the *Box* (□) operator, meaning that after the path of $[\tau^\star\alpha]$, formula φ should hold. Its equivalent ACTL formula is $\neg E[\text{true}_\tau U_\alpha(\neg\varphi)]$.

Property 1: If there is a rising edge on input *P_In*, eventually the output *P_Out* becomes true.

$$AG[\text{P_In!0}][\tau^\star\text{P_Out}\ldots][\tau^\star\text{P_In!1}]A[\text{true}_{true}U_{P_Out!1}\text{true}]$$

A rising edge on *P_In* refers to two clock cycles. In the first cycle it is *0*, then it changes to *1* in the second cycle. The above formula can be read as: for every state in the state space, after a rising edge on *P_In*, *P_Out ! 1* will be eventually reached.

Property 2: Whenever *P_Out* is *1* it becomes *0* in the next cycle; and it remains *0* at least until the next rising edge on *P_In*.

Although there is the explicit expression of *until* in the above property, it cannot be written in one *until* formula because ACTL is unfair. The $\varphi 1 U \varphi 2$ operator in ACTL is known as *strong until*, meaning that the formula is true only if $\varphi 2$ really takes place in a path. Since *P_In* is an input signal, it is possible that it remains at *0* forever. Such behaviour results in *unfair* paths in a model. Unfairness of ACTL means that it cannot express assertions only on fair paths. In other words, a formula has to be analysed on both fair and unfair paths, although the behaviour on unfair paths is not of interest.

Two formulae are used to capture this property. The first says that if *P_Out* is *1* in a clock cycle, then it must be *0* in the next cycle at least until the third clock cycle. The second formula says that if the *P_Out* is *0*, it cannot change to *1* unless *P_ In* changes to *1*.

$$AG[\text{P_Out!1}][\tau^\star\text{P_In}\ldots]A[\text{true}_{\tau\cup P_Out!0}U_{P_In\ldots}\text{true}]$$

The above formula can be read: for every state in the state space, after the path *P_Out ! 1,* $\tau^\star$*P_In*..., there is *P_Out ! 0* or τ until the next *P_in*....

$$AG[\text{P_Out!0}]\neg E[\text{true}_{\neg P_In!1}U_{P_Out!1}\text{true}]$$

The above formula can be read: for every the state in the state space, after *P_Out ! 0*, there does not exist a path such that after actions which are not *P_In ! 1*, *P_Out ! 1* can be reached.

These two properties are verified to be true by CADP, taking just seconds for each of the formula.

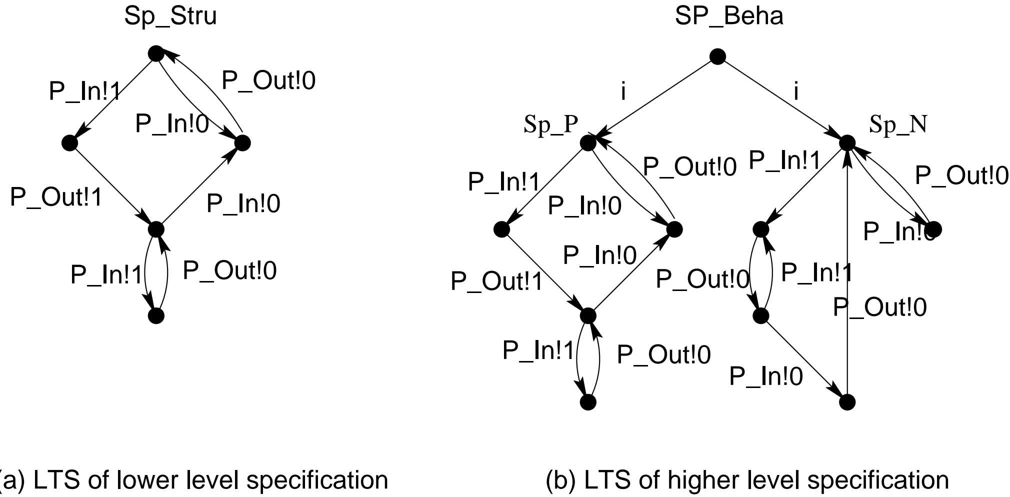


Figure 17: The LTSs of single pulser specifications

4.2.2 Verifying Relations between Two Levels

Most hardware verification tools provide the feature of model checking temporal logics. But circuits specified in DILL can also be analysed by checking the relations between specifications at different abstract levels. This provides an alternative or complement to the temporal logic model checking approach.

Figure 17 shows the LTSs which are observationally equivalent to the DILL specifications. The LTS which is strongly equivalent to the lower level specification has 97 states so it is not feasible to draw it here. Since the higher level specification is more non-deterministic than the lower one, a preorder relation instead of equivalence is considered. CADP can check only two preorder relations: simulation preorder and safety preorder. It is revealed that the lower level specification satisfies the safety preorder but not the simulation preorder when compared with the higher level specifications. Intuitively fulfilling safety preorder can guarantee that all the external behaviour of an implementation is allowed by the specification. Although safety preorder does not in general preserve deadlock and livelock, the implementation has been verified free from them.

To gain more confidence in the circuit, consider the following verification approach. Because it has already been known that the design of the circuit intends to implement a positive edge triggered single pulser, a behavioural specification which deals with the negative edge triggered situation can be extracted from the non-deterministic specification. The state graph of this specification, not surprisingly, is the *Sp_P*, which is part of the left branch of the *Sp_Beha* (figure 17). It is evident that the implementation is observationally equivalent to this deterministic specification. Therefore it is ensured that the circuit is a correct implementation of the deterministic specification.

4.2.3 Case Study

In this section, the DILL approach is evaluated using another benchmark circuit in [SK96], a bus arbiter. The purpose of the Bus Arbiter is to grant access on each clock cycle to a single client among a number of clients requesting the use of a bus. The inputs to the arbiter are a set of request signals, each from a client. The outputs are a set of acknowledge signals, indicating which client is granted access during a clock cycle. The documentation also defines some properties that the Bus Arbiter must respect. They are given informally and also in CTL (Computational Tree Logic). Besides listing the properties to be fulfilled, the benchmark documentation also gives an arbitration algorithm in plain English. Finally the gate level implementation of the Bus Arbiter is provided as a circuit diagram.

Higher-Level Specification in LOTOS

LOTOS supports specification at various levels of abstraction. Although the benchmark circuit has been studied by many researchers, apparently there has not been a formal specification of the arbitration

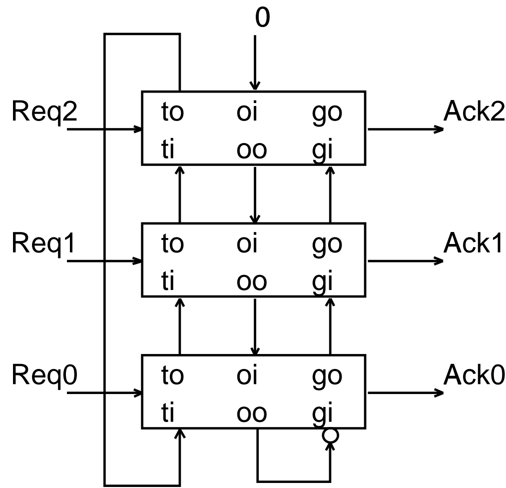


Figure 18: A bus arbiter with three cells

algorithm used in the design. With LOTOS, it is possible to provide such a higher-level specification. There are two clear benefits of this formalization. Firstly, better understanding of the algorithm can be gained from rigorous specification. Secondly, correctness of the algorithm itself can be ensured before the circuit is built and verified. Flaws in the algorithm will be more time-consuming to correct if they are discovered only after the implementation.

The arbitration algorithm embodied in the design is a round-robin token scheme with priority override. Normally the arbiter grants access to the highest priority client: the one with the lowest index number among all the requesting clients. However as requests become more frequent, the arbiter is designed to fall back on a round-robin scheme, so that every requester is eventually acknowledged. This is done by circulating a token in a ring of arbiter cells, with one cell per client. The token moves once every clock cycle. If a client's request persists for the time it takes for the token to make a complete circuit, that client is granted immediate access to the bus.

Translating the algorithm to LOTOS is quite straightforward. It is realized mainly by LOTOS value expressions. For example each cell has two variables associated with it: *token* that indicates if the token is in the cell, and *waiting* that indicates if the request of the corresponding client has persisted for a completed token cycle. Circulating the token, (re)setting the waiting variable and so on correspond to LOTOS value expressions. For an arbiter with three cells, the LOTOS specification has 79 lines (including comments) for the behavioural specification.

Lower-Level Specification in DILL

The design of the arbiter consists of repeated cells. Each cell is in charge of accepting request signals from a client, and sending back acknowledgments to the same client. Figure 18 shows an arbiter with three cells. Figure 19 shows the design of each cell. The first cell is slightly different because it is assumed that the token is initially in the first cell.

The principle of the circuit will not be explained in detail here. Briefly, the *ti* (token in) and *to* (token out) signals are for circulation of the token. The *to* output of the last cell is connected to the *ti* input of the first cell to form a token ring. The *gi* (grant in) and *go* (grant out) signals are related to priority. The grant of cell *i* is passed to cell *i+1*, and indicates that no client of index less than or equal to *i* is requesting. Hence a cell may assert its acknowledge output if its grant input is asserted. The *oi* (override in) and *oo* (override out) signals are used to override the priority.

Because the components of each cell are in the DILL library, it is very easy to specify the process describing a cell. The specification of an arbiter with three cells is obtained by connecting three such processes. As for the Single Pulser, there is also an environment constraint in the structural specification to meet the conditions of the synchronous circuit model discussed in section 3.3.

After both levels have been specified, it is time to verify the circuit. In the following section, all the

Figure 19: Design of an arbiter cell

three verification tasks mentioned in section 4.1.1 will be carried out. That is to say, the specification and implementation are checked against several properties, and the equivalence between the two levels is also examined.

Verification

The benchmark documentation provides three properties the bus arbiter should fulfil. The following three formulae refer to client 0; the formulae for other clients have a similar form. Similar to the *weak box* formula in the example of the single pulser, $< \tau^{\star}\alpha > \varphi$ is the *weak diamond* operator, the shorthand of $E[\text{true}_{\tau} U_{\alpha} \varphi]$.

Property 1: No two acknowledge outputs are asserted in the same clock cycle (safety).

$$AG\neg(EX_{Ack1!1}(< \tau^{\star}\text{Ack0!1} > \text{true} \cup \ < \tau^{\star}\text{Ack2 ! 1} > \text{true}))$$

This formula is read: for every state in the state spaces, it is not the case that there exists an action *A0 !1* which can be followed by *Ack1 !1* or *Ack2 !1*.

Property 2: Every persistent request is eventually acknowledged (liveness).

$$AG([\text{Req0 ! 1}]A[\text{true}_{true} U_{Req0!0 \cup Ack0!1} \text{true}])$$

This formula can be read: for every state in the state space, after the action *Req0 !1*, eventually *Ack0 !1* will be reached, unless there is *Req0 !0*.

Property 3: Acknowledge is not asserted without request (safety).

$$AG([\text{Req0 ! 0}]\neg E[\text{true}_{\neg Req0!1} U_{Ack0!1} \text{true}])$$

This formula can be read: for every state in the state space, after the action *Req0 ! 0*, it is not the case that there exists a path such that after an action which is not *Req ! 1*, *Ack0 ! 1* can be reached.

To verify the higher-level specification against the temporal logic formulae, the LTS of the specification was produced first. Cæsar generates an LTS with 3649 states and 7918 transitions. Aldébaran reduces this to 379 states and 828 transitions with respect to strong bisimulation. Both generation and reduction take seconds. The temporal logic formulae are then checked against the minimised LTS. Each is verified to be true within 1 minute.

The real challenge comes when the lower-level DILL specification is verified. The state space is so large that direct generation of the LTS from the LOTOS specification is impractical. As mentioned before, there are several advanced techniques implemented in CADP to tackle the problem of state space explosion. Nevertheless, using on-the-fly verification of the arbiter also fails after considerable run-time. CADP does

Signal	Cycle1	Cycle2	Cycle3	Cycle4
Req0	1	1	1	0
Req1	0	0	0	1
Req2	0	0	0	0
Ack0	1	1	1	
Ack1	0	0	0	**0** or **1**
Ack2	0	0	0	

Table 1: A counter-example generated by Aldébaran

not currently support the direct generation of BDDs from LOTOS specifications, instead BDDs are only used in several algorithms after the LTSs of LOTOS specifications are obtained.

Compositional generation was tried out to verify the arbiter. Basically the idea is that of `divide and conquer'. A LOTOS specification is divided into several smaller specifications to make sure that it is possible for Cæsar to generate an LTS for each of them. Then Aldébaran is used to reduce these LTSs with respect to a suitable equivalence relation. The minimised LTSs are then combined using the LOTOS parallel operator (and also the **hide** operator if necessary) to form a network of communicating LTSs (the CADP term). At this stage, an LTS might be produced from the network, or on-the-fly verification might be performed against the network. In order to get valid verification results, special attention must be given to the equivalence relation that is used. The relation must be a congruence with respect to the compositional operators, here the LOTOS parallel and hide operators. The relation must also preserve the properties to be verified. This ensures that the resulting network of communicating LTSs will respect the same properties as the original LOTOS specification.

Among the three properties proposed for the benchmark, the first and the third are safety properties while the second is a liveness property. Safety equivalence [BFG+91] preserves all safety properties, while branching bisimulation equivalence [vW89] preserves liveness properties when there is no livelock in specifications. Both of these equivalences are congruences with respect to the parallel and hide operators. These two equivalences are thus appropriate to compositional generation.

The design of the arbiter was divided into three pieces, one per cell of the arbiter. After about seven minutes in total, an LTS which is safety equivalent to the LOTOS specification of the design was generated. The two safety properties were verified to be true against this LTS, implying that the design also satisfies these safety properties. Verification of the formulae takes just seconds. However generating the LTS which is branching equivalent to the design takes a much longer time. To tackle the problem, an environment process which restricts the order of input signals *Req1, Req2, Req3* is applied to both levels of specifications. This helps to reduce the state spaces dramatically. It makes it possible to verify that the implementation is free from livelock and also satisfies the liveness property.

Observational equivalence is chosen for the implementation verification. As before, compositional generation was exploited to generate the LTS for the design. This time each cell was reduced with respect to observational equivalence since it is a congruence for the parallel and hide operators. After about eight minutes in total, the LTS was generated. It was expected that this LTS would be observationally equivalent to the one representing the higher-level specification. However Aldébaran discovered that they are not! Table 1 is one of the sequences given as a counter-example. (The Aldébaran output has been rendered more readable here.) This sequence indicates that in the first three clock cycles only client 0 requests the bus; both the high-level specification and the low-level design grant access to this client. In the fourth cycle, client 0 cancels its request but client 1 begins to request access. At this point the two levels of specifications are different: the lower-level specification offers 0 for *Ack1*, whereas the higher-level specification offers 1 for *Ack1*.

After step-by-step simulation of the counter-example, it was soon discovered that the circuit does not properly reset the *oo* (override out) signal to *0* in the following situation. Suppose a cell has been requesting access, so its *W* register is set to *1*. However the cell cancels the request in the very clock cycle that the token happens to arrive. In this situation, because the client has already cancelled its request it should be

possible for another client to get the bus. However, the design sets the *oo* signal to override the priority as if this client were still requesting. This prevents any other client from accessing the bus in this clock cycle.

Fixing the problem was much easier than finding it. The correction was to connect the *Req* signal to the *And* gate that follows the *W* register. The output of the *And* gate guarantees that the *oo* signal is always correctly set or reset according to the request signal in the current clock cycle. This modified design was verified to be observationally equivalent to the higher-level algorithmic specification.

As mentioned in section 3.3, in DILL the inputs are assumed to be synchronized with the clock signal. If the *Req* signal in figure 19 is not synchronized with the clock. In this case the problem discussed above might not happen. As the benchmark documentation does not state if inputs are synchronized with the clock or not, it is believed that the modified design is more robust.

4.3 Verification of Asynchronous Circuits

Although the method of verifying asynchronous circuits bears many similarities to that of synchronous circuits, there are some differences due to the nature of asynchronous circuits and the modelling techniques adopted by DILL. This section focuses on the differences rather than the similarities

4.3.1 Extra Considerations for Verifying Asynchronous Circuits

The main difference in modelling synchronous and asynchronous circuits is how to represent digital signals. In asynchronous circuits, LOTOS events model physical signal transitions. As has been discussed in chapter 3, the consequence of modelling signal transitions is that the behaviour of structural specifications may not model asynchronous circuits faithfully if the components in the specifications are not specified in the input receptive way. In other words, the behaviour of such specifications is only the subset of all possible behaviour that a real circuit may exhibit. To solve the problem, input quasi-receptive specifications are proposed in section 3.4.6.

So far, only those asynchronous circuits which assume unbounded delay models are specified in DILL. In particular, DILL specifications have a direct mapping to speed-independent circuits. Recall that speed-independent circuits assume zero delay on wires and unbounded delay on components. Therefore the correctness of this kind of circuit is irrespective of the delay magnitudes of components. In practice, speed-independent circuits are regarded as the same as semi-modular circuits, where no input should pre-empt pending outputs. To pinpoint if a circuit is speed independent or not, each of its components should be specified according to the requirement of semi-modularity, i.e. when an input can potentially pre-empt one of its pending outputs, the input leads to the deadlock **stop**, indicating that erroneous behaviour occurs.

The correctness of asynchronous circuits is very sensitive to their environment. Suppose one wants to know if a *Repeater* can be implemented by two *Inverter* gates in series. A straightforward way of verifying the idea is to write both the specification and the implementation of the repeater, and then compare their state graphs. There have been several models of logic gates developed in chapter 3, and many equivalence relations have been discussed in section 4.1.3. But whichever gate model is used, it is discovered that the implementation has more behaviour than the specification; in particular it can receive more inputs than its specification does. As shown in figure 20. The *Inverter* in the figure is modelled according to the gate model in section 3.2, which assumes that pending outputs can be pre-empted. The two level specifications are not equivalent even in terms of traces. In fact, $trace(S) \subset trace(I)$, i.e. they are related by trace preorder. Intuitively this relation means that the implementation can do what the specification dictates, but it can also do what is not given in the specification. In general this relation is too weak to be a good criterion.

It is a very common phenomenon that a structural implementation has much more behaviour than the corresponding behaviour of its specification. This does not just happen in the DILL approach, but also happens in many other methodologies such as those based on process algebra and trace theory. The phenomenon makes it unrealistic to set the equivalence between specifications and implementations as the correctness criterion. In fact an equivalence relation between two LTSs requires that they have the same behaviour under all possible environments. This requirement is usually too strong since practical circuits only operate in some expected environments. In most cases an implementation is only required to be correct in these assumed environments.

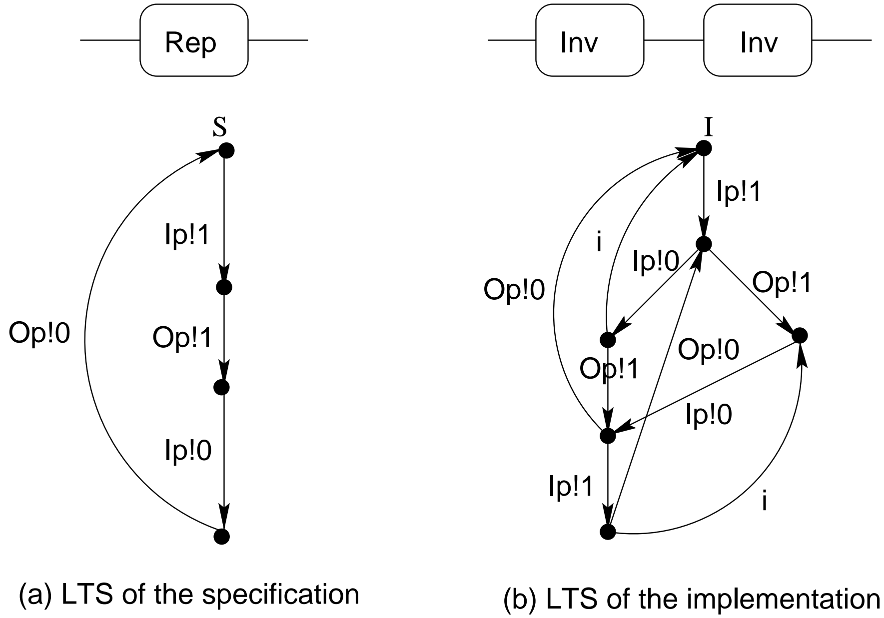


Figure 20: LTSs of the Specification and implementation of *Repeater*

The DILL modelling technique for synchronous circuits assumes a *natural* environment for all synchronous circuits. The environment provides all inputs once during each clock cycle and it provides clock cycle slowly enough to let all outputs settle down. Asynchronous circuits, on the other hand, can accept any inputs at any time, so it is impossible to provide a unified environment for all circuits.

The features of asynchronous circuits have some implications for verification. As it is more difficult to specify components in an input receptive or input quasi-receptive manner, verification of asynchronous circuits may still be based on using components which are not input-receptive. But one should aware that the result of the verification may not exact; in particular the result could be over optimistic in the sense that some bugs cannot be discovered. Using components which are input quasi-receptive, on the other hand, will result in a larger state space and thus make verification more difficult. Since for most asynchronous circuits no explicit environment is given, assumptions about environments have to be made. The next section elaborates this point.

4.3.2 Environment of Asynchronous Circuits

When an environment is not explicitly given, following the approach adopted by David Dill [Dil89] many methodologies simply assume that the mirror of a specifications is the environment of its implementations. The mirror of a specification *S* has the same behaviour as *S*, but its inputs are the outputs of *S* and its outputs are the inputs of *S*. Moreover, it tends to be the most liberal environment an implementation can expect. The reason behind this is that a behavioural specification actually indicates the environment of the circuits. For instance, the specification of the *Repeater* expects its environment first to provide *Ip*, then waits until *Op* has been produced by the circuit. David Dill's idea has been applied in approaches based on trace theory and process algebra, such as in [ESB95, Gop92]. If the idea is to be adopted in the DILL approach, since there is no difference between inputs and outputs in LOTOS, the mirror of the specification is the specification itself. Suppose *S* stands for the specification and *I* represents the implementation, the verification task then becomes comparing $S \parallel I$ with *S* (because $S \parallel S$ is still *S*), or checking if a logic formula holds on $S \parallel I$.

But verifying $S \parallel I$ is not always satisfactory. $S \parallel I$ represents the joint behaviour of *S* and *I*. When an implementation can accept more inputs than its specification does, $S \parallel I$ restricts the considered inputs only to those specified in the specification. This actually assumes that the environment does not provide extra inputs, so the inputs which are only accepted by the implementation are ignored when verifying the

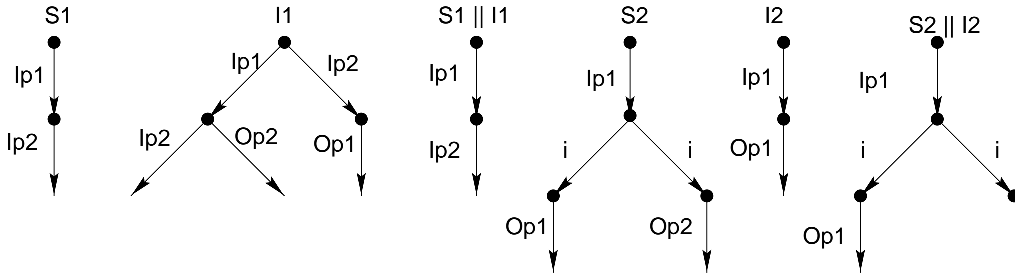


Figure 21: Specification vs Environment

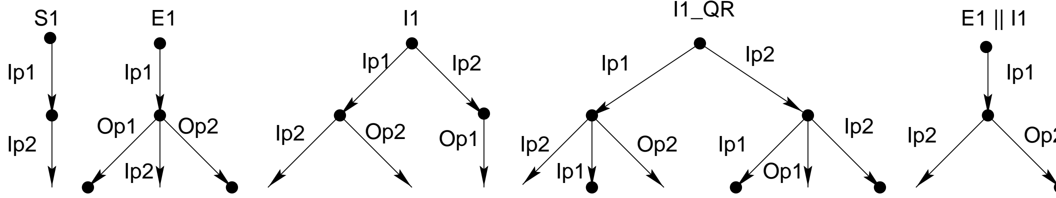


Figure 22: Input quasi-receptive environment

joint behaviour. This is reasonable. But it also permits an implementation to produce more outputs than a specification, since the extra outputs produced by the implementation may also be filtered out. This is however not reasonable since under legitimate inputs, an implementation producing an unexpected output is normally regarded as erroneous. Moreover, when a specification is non-deterministic, this method may exclude correct deterministic implementation. Figure 21 illustrates the intuitive effect of composing a specification with its implementation. *S1* and *I1* are the specification and the implementation of a circuit respectively. If *S1* is regarded as the environment of *I1*, then *S1* || *I1* is the process to be verified. It can be seen that the *Ip2* branch will be ignored during verification. So will the *Op2* branch. Although *S1* || *I1* = *S1*, *I1* is normally regarded as an erroneous implementation due to the extra *Op2* transitions. The rest of the LTSs in the figure illustrate the situation where a specification is non-deterministic. *I2* is a correct deterministic implementation of *S2*. But there is deadlock in *S2* || *I2*. *S2* and *S2* || *I2* are only related by trace preorder, but in general this relation is too weak to be used in verification.

The key point here is the different roles of inputs and outputs in digital circuits. An implementation passively accepts inputs so only those inputs available from the environment make sense. At the same time it positively produces outputs, therefore the environment has no influence on the outputs. A LOTOS specification however does not distinguish inputs and outputs. When it is used as the environment, it restricts them equally.

When an implementation is specified in an input receptive or input quasi-receptive way, the difference between inputs and outputs is actually made. If its environment is also receptive, then it is possible to detect the extra outputs produced by an implementation: if an unexpected output is produced, the environment will go to deadlock after receiving it. Figure 22 applies this idea to *S1* and *I1* in figure 21. *E1* is the input quasi-receptive environment obtained from *S1*. *I1_QR* is the input quasi-receptive form of *I1*. As seen, the unexpected output *Op2* can be detected since *E1* || *I1_QR* leads to deadlock after this output.

However, it is very hard to get the input quasi-receptive environment from a behaviour specification, especially when the specification is complicated or contains internal events. The thesis therefore provides an alternative method for verifying asynchronous circuits. In section 4.3.3, relations which take into account the difference between inputs and outputs will be defined. These relations do not require the receptiveness (or quasi-receptiveness) of the environment or the implementation, and are intuitive criteria of correctness of asynchronous circuits.

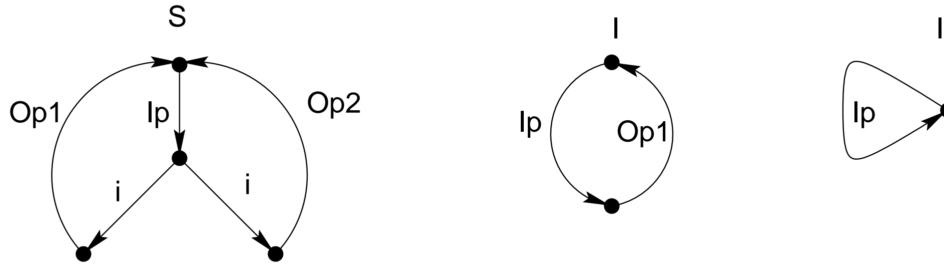


Figure 23: Non-deterministic specification and its implementations

4.3.3 Conformance and Strong Conformance

Although there have been many relations defined to characterise the relationship between two LTSs, they are not very helpful for verifying asynchronous circuits, especially when the environments of a circuit is not explicitly provided. In this section, two new relations, termed *confor* and *strongconfor* are defined to compare an implementation with its specification. These relations take into account the difference of inputs and outputs of a circuit, and are in fact inspired by the *ioco* relation suggested by Jan Tretmans [Tre96] (University of Twente) for testing communication protocols. In chapter 5, more details about *ioco* will be given.

Suppose a circuit has input set L_I and output set L_U. *Spec* and *Impl* are the specification and the implementation of the circuit respectively. *Spec* may be partial in the sense that in some states it does not accept some inputs, i.e. it is not input receptive. As discussed in chapter 3, an input is absent in a state of a specification if the environment of the circuit does not provide this input, if the behaviour of the circuit upon receiving the input is not of interest, or if the behaviour is undefined. Although all circuits are potentially able to accept all their inputs at any time, most specifications are partial to avoid getting into too much details. The implementation *Impl* may either be partial or total in the sense of input receptiveness.

Suppose *sp* is a state of *Spec*, and the corresponding state in *Impl* is *im*. To define the *confor* relation, first consider the input transitions that *sp* and *im* can engage in. For convenience, it is also said that they are the inputs which *sp* or *im* can accept. If an input *ip* is acceptable in *sp*, it means that the environment may provide this input in that state. Therefore it is reasonable to require that *ip* is also accepted in *im*, for otherwise the behaviour of the implementation upon receiving the input will be undefined. On the other hand, if *im* can accept an input which is not acceptable by *sp*, this input and all the behaviour afterwards can be ignored. Since the environment will never provide such an input, or even if the input is provided, such behaviour is not of interest. In short, the input set acceptable in *sp* should be a subset of that acceptable in *im*.

As far as outputs are concerned, intuitively if *sp* can produce *op*, it is expected that a correct implementation should also be able to produce it. If *sp* cannot produce a certain output, neither should its implementation. However, when a specification is allowed to be non-deterministic, requiring that *im* produces exactly the same outputs as *sp* does tends to be too strong, since any deterministic implementation is only able to produce a subset of the outputs dictated by its non-deterministic specification. In this case, a suitable relation should allow output inclusion instead of output equality. The problem is that since an empty set is included in any set, a circuit which `accepts everything but does nothing' may also be qualified as a correct implementation, as shown in figure 23, where both *I* and *I′* are regarded as correct implementations of *S*. To overcome this weakness, a special action δ, which is neither in L_I nor in L_U, is introduced to indicate the absence of outputs. δ is seen as an output action and, like any other output action, if δ belongs to the output set of *im*, it must be in the output set of *sp* for the relation to hold. In other words, *im* can produce nothing only if *sp* can do so. When the δ is considered, *I′* is no longer a legitimate implementation of *S*.

In the above discussion, state *im* is compared with *sp*. *sp* and *im* are not the states in the LTSs of the specification and the implementation, but all the possible situations that the circuits may be in after a certain input-output sequence. Because δ is also involved in the sequence, the state spaces of both specification and implementation are transformed into automata which are explicitly labelled with δ. The

input-output sequences are actually the traces of the automaton of the specification. If the automaton of the implementation cannot follow the sequence, i.e. the sequence is not the trace of the automaton, *im* will be the empty set.

After discussing the informal meaning of the relation, the rest of the section gives the formal definitions. Definition 4.7, definition 4.8 and definition 4.10 comes from [Tre96].

Definition 4.7 (Quiescent Trace and Quiescent State)
Let $p \in \mathcal{LTS}\ (L_I \cup L_U), L_I \cap L_U = \emptyset$

- *A state s of p is* **quiescent***, denoted by* $\delta(s)$*, if* $\forall \mu \in L_U \cup \{\tau\} : s \not\xrightarrow{\mu}$
- *A* **quiescent trace** *of p is a trace* σ *which may lead to a quiescent state:* $\exists p' \in (p\ \mathit{after}\ \sigma) : \delta(p')$
- $out(s) =_{def} \{x \in L_U \mid s \xrightarrow{x}\} \cup \{\delta \mid \delta(s)\}$
- $out(S) =_{def} \bigcup\{out(s) \mid s \in S\}$
- $in(s) =_{def} \{x \in L_I \mid s \xrightarrow{x}\}$
- $in(S) =_{def} \bigcup\{in(s) \mid s \in S\}$

A quiescent state is one that cannot perform any output transition or an internal transition. *out(s)* defines all the output actions that a state can produce. This includes the quiescent `action' δ which means the state cannot produce any output. *in(s)* defines all the input actions that a state can accept.

Definition 4.8 (Suspension Trace)
Let $p \in \mathcal{LTS}(\mathcal{L_I} \cup \mathcal{L_U}), \mathcal{L_I} \cap \mathcal{L_U} = \emptyset$

- $p \xrightarrow{L_U} p' =_{def} p = p'$ *and* $\forall \mu \in L_U \cup \{\tau\} : p \not\xrightarrow{\mu}$
- *The suspension trace of p are: Straces(p)* $=_{def} \{\varphi \in (L \cup L_U)^\star \mid p \xRightarrow{\varphi}\}$

To define suspension traces, the transition relation $\rightarrow$ is extended with the refusal of output actions: self-loop transitions labelled with L_U expressing that no action in the output set can occur. The refusal of output actions can also be expressed by δ transitions. A suspension trace, consequently, not only contains ordinary actions, but also δs. If L_δ denotes $L \cup \delta$, then a suspension trace $\sigma \in L_\delta^\star$.

Definition 4.9 (Conformance and Strong Conformance)
Let $i \in \mathcal{LTS}(\mathcal{L_I}, \mathcal{L_U})$*, and* $s \in \mathcal{LTS}(\mathcal{L_I} \cup \mathcal{L_U}), \mathcal{L_I} \cap \mathcal{L_U} = \emptyset$ *then*

$$i \text{ confor } s \quad =_{def} \quad \begin{array}{l} \forall \sigma \in \text{Straces(s)} : \text{ out}(i \text{ after } \sigma) \subseteq \text{out}(s \text{ after } \sigma) \text{ and} \\ \text{if i after}\, \sigma \neq \emptyset : \text{ in}(s\, \text{after}\, \sigma) \subseteq \text{in}(i \text{after}\, \sigma) \end{array}$$

$$i \text{ strongconfor } s \quad =_{def} \quad \begin{array}{l} \forall \sigma \in \text{Straces(s)} : \text{ out}(i \text{ after } \sigma) = \text{out}(s \text{ after } \sigma) \text{ and} \\ \text{if i after}\, \sigma \neq \emptyset : \text{ in}(s\, \text{after}\, \sigma) \subseteq \text{in}(i \text{after}\, \sigma) \end{array}$$

As will be seen in chapter 5, *confor* is quite similar to the *ioco* relation except that *ioco* assumes the input receptiveness of implementations, so input inclusion is always satisfied.

The *confor* relation requires that after a suspension trace of *s*, the outputs that an implementation *i* can produce are included in what *s* can produce, and if *i* can follow the suspension trace, the inputs that *s* can accept are also accepted by *i*. *strongconfor* has the similar definition except that output inclusion is replaced by output equality.

The *confor* and *strongconfor* relations are more easily observed if the LTSs of specifications are transformed to *suspension automata*, where δ is explicitly labelled.

Definition 4.10 (Suspension Automaton) *Let L be partitioned into* L_I *and* L_U*, and let* $p =< S, L, T, s0 > \in \mathcal{LTS}\ (L)$ *be a labelled transition system;* $\mathcal{P}(\mathcal{S})$ *denotes the powerset of S, i.e. the set of the subsets of S; then the suspension automaton of p,* Γ_p*, is the labelled transition system* $\langle S_\delta, L_\delta, T_\delta, q_0 \rangle \in \mathcal{LTS}(L_\delta)$*, where*

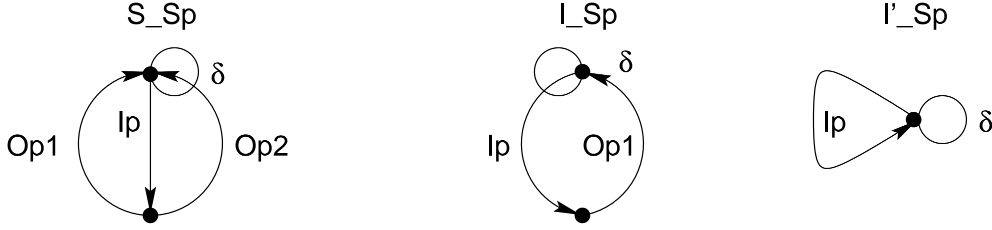


Figure 24: The suspension automata of LTSs in figure 23

$$
\begin{array}{lll}
S_\delta & =_{def} & P(S)\backslash\{\emptyset\} \\
T_\delta & =_{def} & \{q \xrightarrow{a} q' \mid a \in L_I \cup L_U, q, q' \in S_\delta,\ q' = \{s' \in S \mid \exists s \in q : s \stackrel{a}{\Longrightarrow} s'\}\} \\
 & & \cup\{q \xrightarrow{\delta} q' \mid q, q' \in S_\delta, q' = \{s \in q \mid \delta(s)\}\} \\
q_0 & =_{def} & \{s' \in S \mid s_0 \stackrel{\epsilon}{\Longrightarrow} s'\}
\end{array}
$$

An important property of a suspension automaton is that it is deterministic and the suspension traces of a system p coincide with the traces of its suspension automaton Γ_p (the proof can be found in [Tre96]). Moreover, for all $\sigma \in L^\star$, $out(\Gamma_p \text{ after } \sigma) = out(p \text{ after } \sigma)$ and $in(\Gamma_p \text{ after } \sigma) = in(p \text{ after } \sigma)$. Therefore checking the *confor* and *strongconfor* relations can be easily reduced to checking the trace inclusion relation on the suspension automata. The key to generating a suspension automaton from an LTS is to build the transition relation T_δ. From the definition, the first term of T_δ determinises LTS, and the second term adds δ transitions to the states S_δ which contains a quiescent state of *S*.

As an example, the suspension automata of the LTSs in figure 23 are illustrated in figure 24. Since the LTSs are deterministic, their suspension automata are almost identical to themselves, except for the δ transitions. As can be seen, *out(S_Sp after Ip)* = $\{$*Op1, Op2*$\}$, *out(I_Sp after Ip)* = $\{$*Op1*$\}$, and *out(I' _Sp after Ip)* = $\{\delta\}$, therefore *I confor S* but **not** *I' confor S*. There is also **not** *I strongconfor S*. Normally *confor* is used when a specification and an implementation are deterministic, and *strongconfor* is used when an implementation is less non-determinisitic than a specification.

A verifier *VeriConf* developed by the author which checks the *confor* and *strongconfor* has been implemented in the C language. This verifier was developed using the programming interface of CADP. Briefly, CADP is exploited to generate LTSs of both specification and implementation. Then the verifier is used to produce the suspension automata from the LTSs and to compare the automata according to the relations. This verifier has been successfully used in verifying several asynchronous circuits, including the two examples given in the next section.

4.3.4 Case Studies

4.3.5 Asynchronous FIFO

In chapter 3, an asynchronous first-in-first-out buffer was specified. Designed for dual-rail datapaths, this buffer has two inputs *InT, InF* and two outputs *OutT, OutF*. It is assumed to be empty initially. When *1* appears on *InT* or *OutT*, the data on the datapath is *1*. When *1* appears on *InF* or *OutT*, the data is *0*. Lines should be reset to *0* between two transmissions. The specification will not be repeated here. *Spec* is the behavioural specification of a FIFO with two stages. In the following, *TwoStages* represents the implementation using components which are not input receptive, *EnvF, EnvB* is the environment of *TwoStages*, *EnvF_QR* and *EnvB_QR* are the environment specified in input quasi-receptive manner, and *TwoStages_QR* is the implementation which replaces all the components in *TwoStages* with their corresponding input quasi-receptive components. Note that *EnvF_QR* is actually identical with *EnvF* since *EnvF* has no input. For this circuit, the following verification methods are applied:

- The liveness property is specified in ACTL and it is verified that the specification satisfies the following property.

Property: If there is an input data *1*, then outupt will become *1* eventually:

$$AG([\text{InT ! 1}]A[\text{true}_{true}U_{Out!1}\text{true}])$$

The formula for data *0* is similar and is also verified to be true.

- In this example, the environment is explicitly given. It was verified that *Spec* $\approx$ *TwoStages* || *(EnvB* | $[\cdots]$ | *EnvF)*. Recall that $\approx$ denotes the observational equivalence.
- For checking speed-independence, input quasi-receptive components are used. It was also verified that *Spec* $\approx$ *TwoStages_QR* || *(EnvB_QR* | $[\cdots]$ | *EnvF_QR)*, which gives more confidence in the design of the FIFO.
- The implementation *TwoStages_QR* || *(EnvB_QR* | $[\cdots]$ | *EnvF_QR)* also satisfies the liveness property.
- Using *VeriConf*, it was established that *TwoStages_QR* || *(EnvB_QR* | $[\cdots]$ | *EnvF_QR) strongconf Spec*.

A Circuit with two Components

This circuit is an example in [Ebe91], where it is used to show the difference between speed independence and delay insensitivity. Although small, it reveals the necessity of using input quasi-receptive specifications. Following the specification style of that paper, value offers *! 1* and *! 0* are omitted. As a matter of fact, for asynchronous circuits, value offers are not necessary as long as the initial states of signals are known. For instance, if signal *Ip* is initially *0*, then *Ip,*$\cdots$*, Ip,*$\cdots$*, Ip,*$\cdots$ is the short form of *Ip ! 1,*$\cdots$*, Ip ! 0,*$\cdots$*, Ip ! 1,* $\cdots$. Apparently, keeping the value offer is helpful only for readability.

The behavioural specification is shown in figure 25(a). The behaviour of the two components is shown in figures 25(b) and 25(c) respectively. In fact *Ele1* achieves the *Or* function of signal transitions, and *Ele2* achieves the *And* function. The proposed implementation is in figure 25(c). The verification task is to check if this implementation is speed-independent and delay insensitive. For analysing delay insensitivity, the circuit is transformed to figure 25(e), where the isochronic fork in (d) is replaced by an explicit *fork* element. An alternative transformation is to add two *delay* elements, as explained in figure 11 of section 3.4.2. The two methods have the same effect.

The implementation of the figure 25(d) is specified as *Impl1*, and figure 25(e) is specified as *Impl2*:

```
process Impl1 [IA, IC, ID, OB, OE] : noexit :=
 Ele1 [IA, IC, OB]
|[IC]|
 Ele2 [IC, ID, OE]
endproc

process Impl2 [IA, IC, ID, OB, OE] : noexit :=
hide x, y in
 ( Ele1 [IA, x, OB]
|||
 Ele2 [y, ID, OE])
|[x,y]|
 Fork [IC, x, y]
endproc
```

The state spaces of *Impl1* and *Impl2* are much larger than that of *Spec*. For example, both can accept *IC* and *ID* from their initial states, but *Spec* cannot. Since no explicit environment is given, a direct verification approach is to compare *Impl1* || *Spec* with *Spec*, with the assumption that *Spec* is also the environment of its implementations. According to CADP, they are observationally equivalent. The same result holds for *Impl2*, i.e. *Impl2* || *Spec* $\approx$ *Spec* and *Impl1* || *Spec* $\approx$ *Spec*. This suggests that both figure 25(d) and (e) are correct implementation with respect to *Spec*.

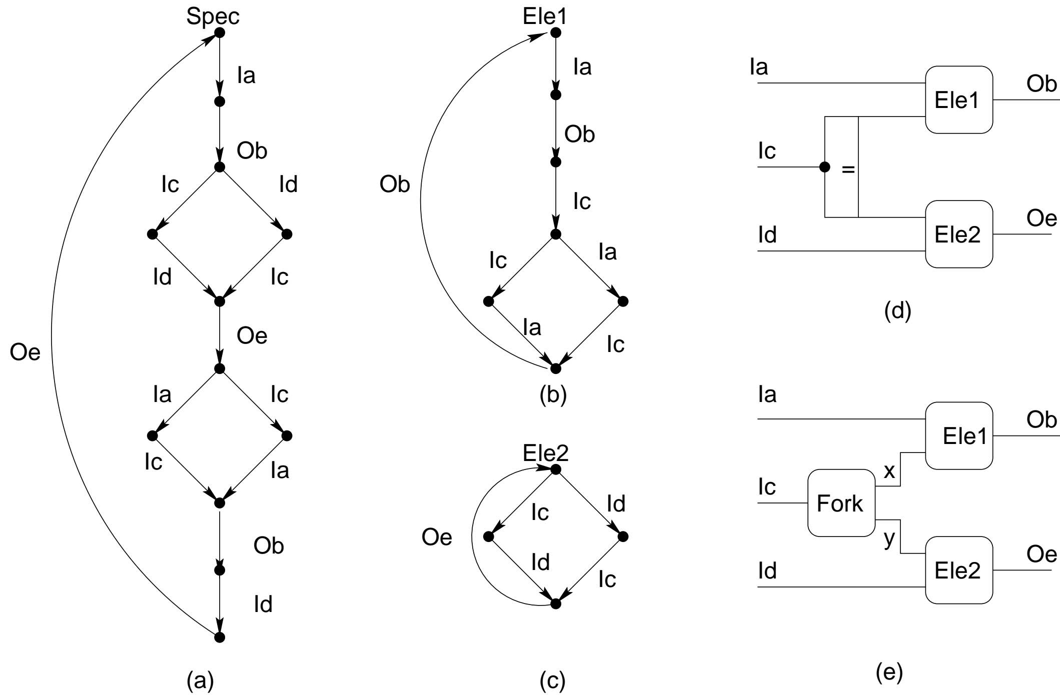


Figure 25: A circuit with two elements

However, to ensure that both of the circuits are really speed-independent implementations of the specification , a more accurate model of the components, i.e the input quasi-receptive model, should be used. Figure 26 shows the LTSs of these input quasi-receptive components. Suppose the implementations are *Impl1_QR* and *Impl2_QR*, using *VeriConf* it is discovered that although there is *Impl1_QR strongconfor Spec*, *Impl2_QR* does not relate to *Spec* with respect to both the *confor* and *strongconfor* relations. A diagnostic trace is given by the verifier: *IA, OB, ID, IC, OE, IC, IA*. By analysing this trace, it can be found that after the *OE* is produced, the *Spec* is able to receive *IA* and *IC*. But for the implementation in figure 25(e), after *OE* is produced the *Fork* element may still be in the unstable state since *x* has not been produced yet. In this unstable state, the *IC* input from the environment makes the behaviour of the *Fork* component undefined, which means that figure 25(e) is not speed-independent. More precisely the correctness of the circuit depends on the speed of *Fork*. Figure 25(d) is therefore not a delay insensitive implementation of *Spec*.

4.4 Related Work

Formal verification of digital circuits has been insensitively investigated during the last decades. There are essentially two approaches to formal verification: model checking and theory proving. Several mature tools have been developed, such as SMV (Symbolic Model Verifier [McM93]), VIS (Verification Interaction with Synthesis [BH$^+$96]), and Cospan [HHK96]. These tools are mainly based on model checking to enable automatic verification. But general purpose theorem provers, such as HOL [Mel93] and PVS [SRC96] are also employed in verifying digital circuits, especially the datapathes.

CADP belongs to model checking tools so is similar to SMV, VIS etc. But unlike these tools which only provide the temporal logic model checking, CADP supports relation checking (equivalence, preorder etc) and therefore provides more verification approaches. It is well known that temporal logic formulae are difficult to write even for experienced users. Many negative verification results are actually because of improper formulae instead of erroneous hardware designs. Relation checking helps to get rid of this problem, and moreover, is able to point out more errors than temporal logic model checking, as has been discovered in the case study of the bus arbiter. The reason is that a higher level specification of a system

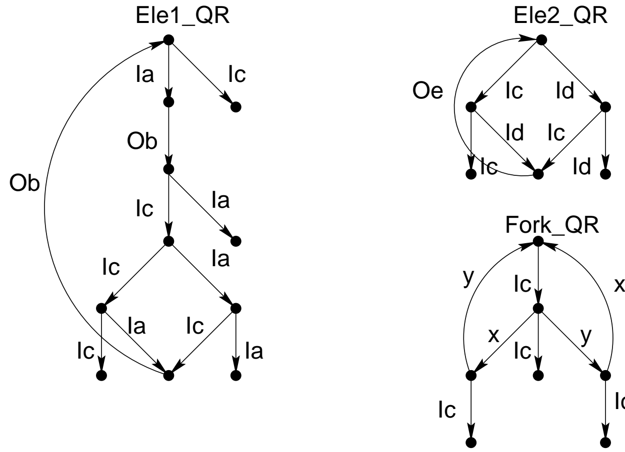


Figure 26: Quasi-receptive specifications of components in figure 25

contains more details than a formula which expresses an abstract property.

The size of the synchronous circuit that can be effectively verified is small compared to that handled by other tools. COSPAN can verify an arbiter with four cells with the consumption of about 1 MB memory, due to a symbolic representation using BDDs and efficient reduction techniques [FK97]. CIRCAL is reported to generate the state space of an arbiter with up to 40 cells using reasonable computing resources, although the actual memory used was not reported [MM93]. Again this is due to the BDD representation of the CIRCAL specification. Note that in fact the arbiter was not formally verified in CIRCAL. [MM93] just gives a test pattern to show that even if all clients request the bus, only one can gain access to the bus in each clock cycle. CADP on the other hand consumes more than 100 MB of memory to produce the state space of a three-cell arbiter. Although the resulting state space is relatively small, the intermediate stages of generation need considerable memory.

The main reason for the performance limitation is that in synchronous circuits the order in which signals occur during a clock cycle is not so important. So it is reasonable to imagine that the inputs happen together and then output occurs. But when modelling such circuits in DILL, independent (interleaved) inputs are allowed so the state space is considerably enlarged. The second reason is related to the verification tool. CADP is perhaps the most mature tool which supports LOTOS, but it is still under development and currently some of its features are mainly based on explicit state exploration. Because CADP cannot produce the minimised state space in the first place, large amounts of memory have to be consumed before a smaller LTS can be produced by minimisation. On-the-fly algorithms are of some help, but they apply only in particular situations. For example, on-the-fly observational equivalence checking is not supported by CADP. Also CADP does not offer a BDD direct representation of LOTOS specifications, although BDDs are used to represent intermediate data types in some algorithms.

The tools mentioned above mainly deal with synchronous circuits. Verification of asynchronous circuits, especially SI and DI circuits is also an active area. Based on trace theory, Dill [Dil89] built a verifier, which could be the first automatic tool of this kind. Other tools can be found in [KKTT98, ESB95]. Most asynchronous tools known to the author are model checking tools, which differ each other only in how the models are represented, or how to design algorithms to improve the verification performance.

Like DILL many other asynchronous verification approaches also define relations to indicate that in what sense a circuit design is regarded as correct. The relations *confor* and *strongconfor* in this thesis resemble the *conformance* in [Dil89], *decomposition* in [ESB95] and *strong conformance* in [GBMN94]. The former two are based on trace theory. They cannot detect deadlocks and livelocks in specifications. The last approach is based on CCS, so it is possible to detect deadlocks and livelocks after the specifications are obtained. But the *strong conformance* requires that an implementation should not produce less outputs than its specification does. This excludes the possibility of applying the relation on non-deterministic specifications. The *confor* and *strongconf* defined in this chapter have clear advantages over these relations. Firstly they give a more intuitive interpretation of the correctness of implementations. Secondly they

consider suspension traces of a specification instead of traces, which makes the relations stronger and is capable to detect more errors. Finally unlike the *conformance* and *strong conformance* relations which are intended to suit all kinds of specifications, *confor* and *strongconfor* are used with non-deterministic and deterministic specifications respectively, which helps to make the verification results more accurate.

4.5 Conclusion

This chapter presents the DILL approach to verifying both synchronous and asynchronous circuits.

DILL supports all the three kinds of hardware verification tasks, namely *requirements capture, implementation verification* and *design verification*. In DILL *requirements capture* and *design verification* are performed by model checking temporal logic formulae, and *implementation verification* is conducted by comparing the relations between LTSs.

Compared to verifying asynchronous circuits, verifying synchronous circuits is more straightforward. The existing LOTOS verification tools can be employed directly so that substantial efforts on developing tools can be saved. This is one of the main reasons that LOTOS is considered to be used as a hardware description language.

For verifying asynchronous circuits, more endeavor is needed. This is mainly because of the gap between the different communication schemes in LOTOS and in digital circuits. In LOTOS the communication between processes is symmetric, but in real hardware the communication between components is asymmetric. In LOTOS an event offer can be refused, but in circuits an input signal transition can never be rejected. The DILL model of synchronous circuits does not suffer from the gap since a LOTOS event does not model a signal transition directly.

Two efforts are made to bridge the gap. One is at the specification stage. For structural specifications, their components are specified in input receptive or input quasi-receptive manner, reflecting that inputs are always acceptable. The second is at the verification stage. New relations *confor* and *strongconfor* are defined to take into account the difference between inputs and outputs. The relations provide an intuitive interpretation of correctness of a circuit implementation, and a verifier *VeriConf* for them has been developed.

5 Testing Digital Logic Designs

This chapter presents the DILL approach to testing designs of digital circuits. In the community of hardware designers, the term *testing* usually refers to the activity of detecting manufacturing defects in physical products. While in the community of formal methods, *testing* is one of the validation methods, which can be applied in various stages of design processes. The chapter interprets testing according to the latter meaning. In this sense an Implementation Under Test (IUT) might either be a physical product, or a formal or informal model.

The chapter first presents the background knowledge, where the two validation methods: verification and testing are compared to each other. The theory of conformance testing for LTSs are also briefly introduced in this first section. Testing theory for IOLTSs, the LTSs which differentiate inputs and outputs, is then elaborated, with the focus on the implementation relations *ioconf* and *ioco*, and corresponding test generation algorithms. Following the introduction of the theory, the chapter applies it in validating synchronous and asynchronous circuits. Several examples are used to illustrates the suitability of the approach. To achieve satisfactory coverage of the test cases generated, an algorithm based on a transition tour of the state space graph is developed and implemented. A testbench is also developed to automate testing processes. Finally a benchmark circuit, the *BlackJack Dealer*, is studied to examine the approach.

5.1 Background

5.1.1 Testing and Verification

Testing is an operational way to check the correctness of a system implementation by means of experimenting with it. Tests are applied to the implementation under test, and, based on observations made during the execution of the tests, a verdict about the correct functioning of the implementation is given.

Compared to verification, testing is a more pragmatic way of checking a system. Although both aim to check the correctness of a system, verification is performed on a mathematical model of the system, while testing is done by operating an executable implementation (either a product or an executable model). Verification is exhaustive and can ensure the correctness of a system being checked, but this sureness only applies to the model of the system. Testing is based on observing only a small subset of all possible behaviour, thus it can never be exhaustive. Unlike verification, testing is normally used to discover errors, not to prove correctness. Testing can be applied to real implementations, so is extremely useful when a valid and reliable model is difficult to build or when the system is too complex to be efficiently verified.

In the previous chapter, circuit designs were verified against their specifications. As has been seen, the state spaces of circuit implementations are considerable larger than that of their specifications. Some circuits have such complex behaviour that their state space cannot be efficiently built, making verification impossible. By means of testing, there is no need to build the state spaces of implementations. Only the formal models of specifications are required, therefore much larger circuits can be effectively analyzed. Moreover, testing can be conducted on more detailed models of implementations, which is helpful for finding subtle bugs which may not be captured by validating a formal model.

5.1.2 Formal Conformance Testing for LTSs

There are many aspects of a system that can be tested. Conformance testing answers the question of `does an implementation conforms to its functional specification?' Other kinds of testing include performance testing (`how fast can an implementation perform its task?'), robustness testing (`how does an implementation react when its environment does not behave as expected?') and so on. This chapter applies the developments in the area of formal conformance testing to validate digital circuits.

Formal conformance testing comprises several ingredients: a formal specification, an implementation under test (IUT), an implementation relation, and a test suite. Preferably there should also be a test generation algorithm which helps to generate test suites automatically. Specifications can be written in a formal language such as SDL [ITU92, ITU95], LOTOS [ISO89], or Estelle [ISO97, Tur93]. An implementation is treated as a black box exhibiting behaviour by interacting with its environment. In order to establish a formal relation between specification and implementation, it is assumed that any implementation has a

formal model which can be reasoned about. The assumption that all implementations have formal models is referred to as *test hypothesis* in some literature [Ber91]. Note that the model is only assumed to exist, but it is not known a priori; the implementation relation is the criterion for judging if an implementation conforms to its specification. Test suites contain the test cases used for experiments. A test suite which distinguishes exactly between all conforming and non-conforming implementations is said to be *complete*. Unfortunately, complete test suites are not always available in practice, as they are usually infinite. A weaker requirement is therefore applied: a test suite should be *sound*, i.e. it gives a negative verdict only when an implementation is incorrect. In other words, all correct implementation and possibly some incorrect implementations will have positive verdicts. Comparatively, a test suite is *exhaustive* if it gives a positive verdict only when an implementation is correct. In other words, all incorrect implementations and possibly some correct implementations will have negative verdicts. The errors detected by a sound test suite are real errors, and the correct implementations qualified by an exhaustive test suite are really correct.

Formal conformance testing for Labelled Transition Systems (LTSs) has been intensively studied. Traditional testing theory for LTSs aims at defining implementation relations, instead of finding test suites to characterise implementations. Given a class of tests, a transition system p is related to a system q if for all possible test cases in the class, the observation made of p is in some sense related to that made of q. Such a definition of implementation relation by explicit use of tests and observations is termed *extensional definition*. Many different relations, including bisimulation, testing preorder/equivalence, failure preorder/equivalence [DH84, Nic87] have been defined in the framework of testing theory. In [Bri88], Brinksma studied the possibility of systematically deriving test cases for some implementation relations from their specifications. In his framework, specifications, implementations and test cases are all modelled as LTSs. He pointed out that it is not known if testing preorder (called *red* relation in that paper) is testable or not (i.e. if there exists a complete test suite for any specifications such that the correctness of an implementation with respect to the implementation relation can be characterised by the test suite). For this reason, Brinksma defined an implementation relation termed *conf*. Informally, an implementation i conforms to a specification s with respect to the *conf* relation, i.e i *conf* s, if "testing the trace of s against i will not lead to unexpected deadlocks that could not occur with same test performed with s" [Bri88]. The relation *conf* ensures that an implementation does what it is required to do, but it does not guarantee that the implementation does not do what it is not allowed to do. It is understood that the latter requirement is the task of robustness testing. An important property of *conf* is that it is testable for any specification S, and that for each S test suites can be derived from S. Several test generation algorithms were developed for *conf* relations [PF90, Wez89, Led92] after the work of Brinksma.

Based on Brinksma's theory, conformance testing for IOLTSs (Input Output Labelled Transition Systems) was proposed by Jan Tretmans [Tre96]. Informally an IOLTS is a special LTS in which all inputs are always enabled in any state. The observation is that most real-life implementations distinguish inputs and outputs of a system. Outputs are actions that are initiated by and under the control of a system, while input actions are initiated by and under the control of the system's environment. A system can never refuse to perform its inputs, and its outputs can never be blocked by environment. In Tretmans' theory, specifications are still modelled as LTSs in order to give abstract representations of systems. Implementations however are assumed to have the model of IOLTS, which is believed to be closer to real-world objects. Several implementation relations are defined from LTS to IOLTS. Tretmans proved that the relations *ioconf* and *ioco* are testable and test cases can be derived from any specifications for these two relations. Moreover, he gave a test generation algorithm which guarantees sound test cases. Because test cases are generated from specifications, they are also modelled as LTSs instead of IOLTSs. Like the relation *conf*, *ioconf* and *ioco* can ensure that an implementation does what it is required, but cannot guarantee it does not do what it is not allowed to do. In section 5.1.5, full detail of IOLTS and the two relations will be presented.

As already pointed out in chapter 3, digital hardware communicates with its environment via inputs and outputs. Thus an IOLTS should be more suitable than an LTS for modelling circuits.

5.1.3 Overview of the Approach

In the DILL approach to testing digital circuit designs, the intended behaviour of a circuit is specified in LOTOS, whose semantics is given by an LTS. The implementation of the same circuit is described by VHDL (VHSIC Hardware Description Language [IEE93]). The behaviour of a VHDL program is

presumed to be modelled by an IOLTS. This model is merely assumed to exist – it need not be known explicitly. Implementation relations *ioconf* and *ioco* are used as the correctness criteria for synchronous and asynchronous circuit designs respectively.

The test suite for a circuit is generated from a LOTOS specification following an algorithm based on that proposed in [Tre96]. CADP has been explored to generate hardware test suites automatically. Each test case in the generated test suite is a sequence of input and output signals. Designing test cases as input-output sequences is close to engineering practice in hardware testing. Moreover, it allows test execution and obtaining test verdicts to be completely automated. This is achieved by a VHDL testbench that executes and evaluates the test cases. If there is an inconsistency between the formal specification and its VHDL implementation, the implementation is regarded as incorrect. Figure 27 outlines this approach.

5.1.4 Conformance Testing for IOLTS

This section introduces the conformance testing theory for IOLTS, with the focus on implementation relations *ioconf* and *ioco*, and their corresponding test generation algorithms.

5.1.5 IOCONF and IOCO

As mentioned earlier, an implementation is assumed to have the model of *IOLTS* in Tretmans's framework.

Definition 5.1 (IOLTS) *An input-output transition system p is a labelled transition system in which the set of actions L is partitioned into input actions L_I and output actions L_U ($L_I \cup L_U = L, L_I \cap L_U = \emptyset$), and for which all input actions are always enabled in any state:*

$$\textit{whenever} \qquad p \stackrel{\sigma}{\Longrightarrow} p' \qquad \textit{then } \forall a \in L_I : p' \stackrel{a}{\Longrightarrow}$$

The class of input-output transition systems with input actions in L_I and output actions in L_U is denoted by $\mathcal{IOTS}\,(L_I, L_U) \subseteq \mathcal{LTS}\,(L_I \cup L_U)$.

From this definition, it can be seen that the action set of an IOLTS is partitioned into disjoint input actions and output actions. Each reachable state of the system can always participate in all the input actions.

Specifications, however, are still modelled as an LTS to have an abstract view of systems. Such specifications are interpreted as incompletely specified input output labelled transition systems. i.e. IOLTSs where a distinction between inputs and outputs is made, but where some inputs are not specified in some states. The intention of incomplete specifications might be for implementation freedom, or because the specifier can ensure that the environment will not provide some inputs.

There are several implementation relations defined from LTSs to IOLTSs. Some of these relations, such as the one analogous to testing preorder, are too strong in that they require that specifications are also IOLTSs for the relation to hold. This is obviously impractical in most cases. Two relations, namely *ioconf* and *ioco*, which are analogous to *conf* mentioned in the previous section, are defined for the purpose of conformance testing.

Definition 5.2 (ioconf) *Let $i \in \mathcal{IOTS}\,(L_I, L_U), s \in \mathcal{LTS}\,(L_I \cup L_U)$, then*

$$i \ \textit{ioconf} \ s \quad =_{def} \quad \forall \sigma \in \textit{traces(s)} : \ \textit{out}(i \ \textit{after} \ \sigma) \subseteq \textit{out}(s \ \textit{after} \ \sigma)$$

Recall that in definition 4.7 (section 4.3.3), *out(s)* is defined as all the output actions that a state *S* can perform, which also includes the quiescent action δ. *ioconf* means an implementation is correct if after all the traces σ of the specification, the outputs which an implementation can produce can also be produced by the specification. Since this also holds for δ, the implementation may not show output only if specification cannot do so. This means that those implementations which `accept anything but do nothing' are not qualified as correct implementation according to *ioconf*. Note that the relation requires only for all the traces of specification *S* that the *out*-set inclusion holds. So it allows an implementation to accept more inputs than a specification does. The implementation may do what it wants after it accepts such unspecified

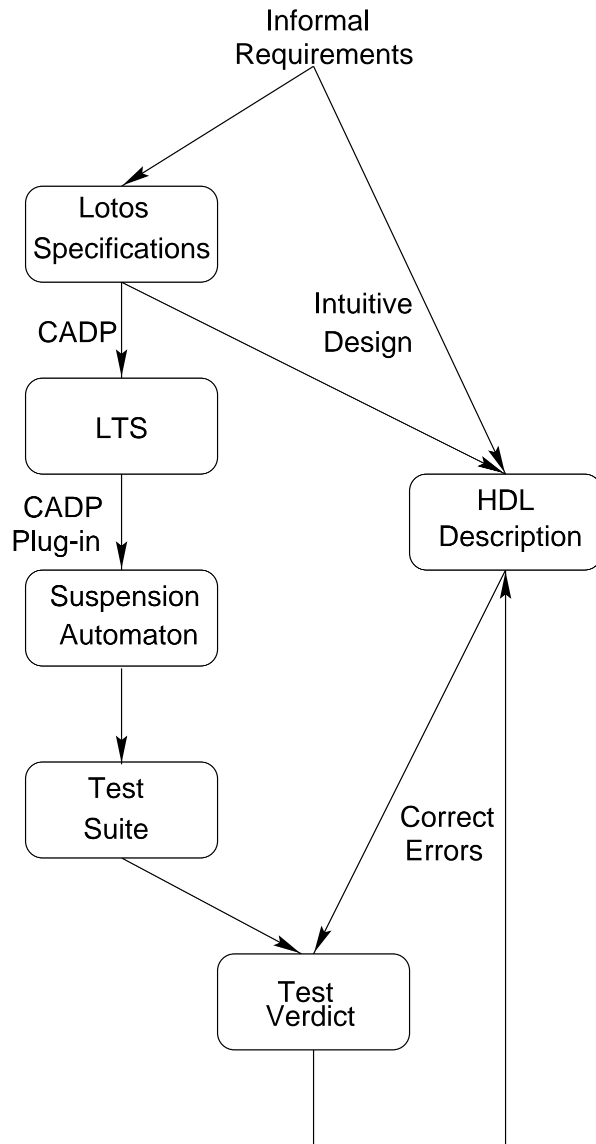


Figure 27: DILL approach of testing circuit designs

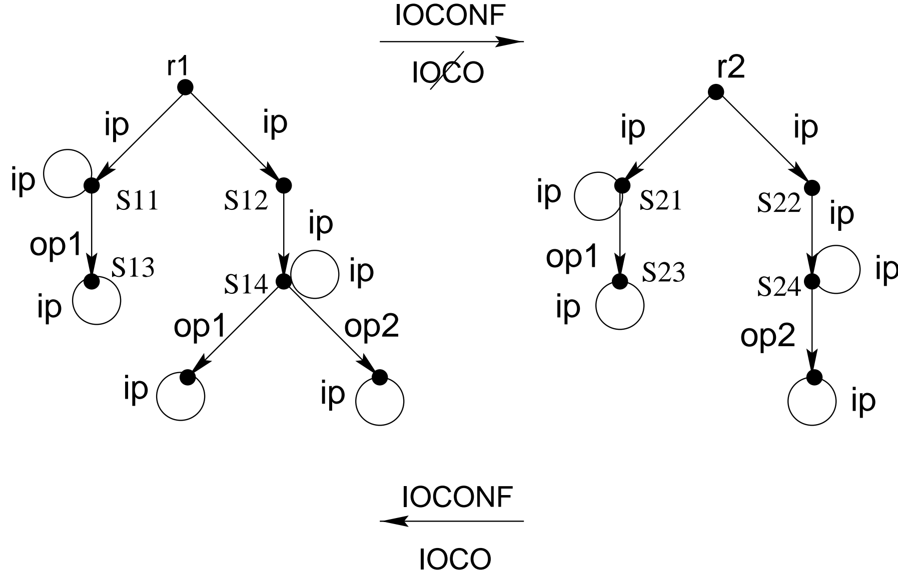


Figure 28: The difference between IOCONF and IOCO

inputs. Therefore similar to the *conf* relation, it just ensures that an implementation does what it is required, but the relation does not force an implementation do not do more than what is required by the specification.

ioco is similar to *ioconf* but is stronger. It restricts inclusion of *out*-sets to suspension traces of specifications.

Definition 5.3 (ioco) *Let* $i \in \mathcal{IOTS}\ (L_I, L_U)$, $s \in \mathcal{LTS}\ (L_I \cup L_U)$, *then*

$$i\ ioco\ s \quad =_{def} \quad \forall \sigma \in \mathit{Straces(s)}:\ out(i\ \ after\ \ \sigma) \subseteq out(s\ \ after\ \ \sigma)$$

In the definition, *Strace* stands for suspension trace, which has been defined in definition 4.8 (section 4.3.3). A suspension trace $\sigma \in L_\delta^\star$ is a sequence of ordinary actions and action δ, the latter is used to denote the absent of output actions. *ioco* is very similar to the relation *confor*, the only difference is that *ioco* assumes that implementations are modelled as IOLTSs which can always accept all inputs, while implementations in *confor* are not necessarily to be input receptive. Consequently, the input inclusion condition, required by *confor*, is always satisfied when *ioco* is considered.

The relations *ioconf* and *ioco* are illustrated in figure 28. These are two IOLTSs *r1* and *r2* with $L_I = \{ip\}$ and $L_U = \{op1, op2\}$. The only difference is at the states *s14* and *s24*; *r1* can produce output *op1* and *op2*, while *r2* can only produce *op2*. *out (r1 after(ip·ip)) = out (*$\{$*s11, s14*$\}$*) =* $\{$*op1, op2*$\}$, and *out (r2 after(ip·ip)) = out (*$\{$*s21, s24*$\}$*) =* $\{$*op1, op2*$\}$. After comparing all the other traces of *r1* and *r2*, it can be concluded that *r1 ioconf r2* and *r2 ioconf r1*. Comparatively, for the relation *ioco* suspension traces are needed. Thus *out (r1 after(ip·*δ*·ip)) = out (*$\{$*s14*$\}$*) =* $\{$*op1, op2*$\}$, while *out (r2 after(ip·*δ*·ip)) = out (*$\{$*s24*$\}$*) =* $\{$*op2*$\}$. That is *r2 ioco r1*, but **not** *r1 ioco r2*. Intuitively, an *ioconf* tester cannot notice that a state is free of output actions when performing testing, but an *ioco* tester can. In this example, after the first *ip* is applied the *ioco* tester may observe that there is no output produced. He then provides another *ip* to see the reaction. *r1* is able to produce either *op1* or *op2* in response, but *r2* can only produce *op2*. However because an *ioconf* tester cannot sense the absent of outputs, he cannot distinguish the behaviour of *ip·ip* and *ip·*δ*·ip*. After providing two concatenated *ip* inputs, he will observe either *op1* or op2 for both systems.

As will be explained later when testing synchronous circuits, δ transitions can be ignored. So *ioconf* will be used for testing synchronous circuits. *ioco* will be explored for testing asynchronous circuits because it has more distinguishing power.

5.1.6 Test Generation

Having seen the models for specifications, implementations and relations between them, it is time to find out the test cases which characterise correct implementations of a given specification with respect to the relations.

For *ioconf* and *ioco* relations, test cases are modelled as $\mathcal{LTS}(\mathcal{L_I} \cup \mathcal{L_U} \cup \{\delta\})$. For practical reasons, they have to have finite behaviour, because any experiment should last for a finite time. In addition, they should be deterministic to allow a tester to have control over test execution. This also requires that test cases have no choice between multiple input actions, nor a choice between input and output actions, as both introduce unnecessary nondeterminism during a test experiment. As a result, a state of a test case is a terminal state, or offers one input to the implementation, or accepts all possible outputs from the implementation (including the δ action). Finally, to be able to decide about the success of a test, the terminal states of a test are labelled with *pass* or *fail*.

Definition 5.4 (test cases and test suites)

- *A test case t is a labelled transition system* $< S, L_I \cup L_U \cup \{\delta\}, T, s0 >$ *such that*
  - *t is deterministic and has finite behaviour;*
  - *S contains the terminal states pass and fail, with init(pass) = init(fail) = $\emptyset$;*
  - *for* $\forall t' \in S$ *of the test case,* $t' \neq$ *pass, fail, either init(t')* $= \{a\}$ *for some* $a \in L_I$*, or init(t')* $= L_U \cup \{\delta\}$.

    The class of test cases over L_U *and* L_I *is denoted as* $\mathcal{TEST}(L_U, L_I)$.
- *A test suite T is a set of test cases:* $T \subseteq \mathcal{TEST}(L_U, L_I)$.

Recall that *init(s)* denotes all the actions in which state s can engage, including the initial transition **i** (definition 2.2). Note that L_I and L_U refer to the inputs and outputs from the point of view of the implementation under test, so L_I is the outputs, and L_U is the inputs of test cases.

When an implementation is tested by a test (called a test run), it will only stop (i.e. deadlock) at the *pass* or *fail* states. Since for other states, either they can offer an input action, in which case the implementation can always accept it, or they offer all output actions, including a δ transition, in which case at least δ can be accepted by an implementation. If deadlock happens at *pass* state, it is said that the implementation passes the test run. Since an implementation can be nondeterministic, different terminal states can be reached with different test runs of the same test case. Only when an implementation passes all possible test runs, is it said that the implementation passes the test case.

To facilitate the generation of test cases, *a suspension automaton* (definition 4.10 in section 4.3.3) of the specification LTS is first built; test generation algorithm is then applied on the automaton.

Recall that a suspension automaton Γ_p of an LTS p is obtained by determinizing p and adding necessary δ transitions. The suspension traces of p coincide with the traces of its suspension automaton Γ_p. In addition, for all $\sigma \in L^\star$, $\text{out}(\Gamma_p \text{ after } \sigma) = \text{out}(p \text{ after } \sigma)$. Therefore checking *ioconf* and *ioco* can be easily reduced to checking trace inclusion relation on suspension automata. For *ioco*, all the traces of suspension automaton should be used for checking, while for *ioconf* only traces without δ transitions are checked.

Definition 5.5 (Test generation algorithm) *Let* Γ *be the suspension automaton of a LTS s, and let* $\mathcal{F} =$ *trace(*Γ*) for the case of ioco and* $\mathcal{F} = \{\sigma \in L^\star \mid \sigma \in trace(\Gamma)\}$ *for the case of ioconf, then a test case* $t \in \mathcal{TEST}(L_U, L_I)$ *is obtained by a finite number of recursive applications of one of the following three nondeterministic choices:*

1. *(* terminate the test case *)*

$$t := pass$$

2. *(* give a next input to the implementation *)*

$$t := a; t'$$

   where $a \in L_I$*, such that* $\mathcal{F}' = \{\sigma \in L_\delta^\star \mid a \cdot \sigma \in \mathcal{F}\} \neq \emptyset$*, and t' is obtained by recursively applying the algorithm for* $\mathcal{F}'$ *and* Γ'*, with* $\Gamma \xrightarrow{a} \Gamma'$.

3. *(* check the next outputs of the implementation *)*

$$
\begin{aligned}
t \quad := \quad & \sum \{x; \mathit{fail} \mid x \in L_U \cup \{\delta\}, x \notin \mathit{out}(\Gamma), \epsilon \in \mathcal{F}\} && (1) \\
[] \quad & \sum \{x; \mathit{pass} \mid x \in L_U \cup \{\delta\}, x \notin \mathit{out}(\Gamma), \epsilon \notin \mathcal{F}\} && (2) \\
[] \quad & \sum \{x; t_x \mid x \in L_U \cup \{\delta\}, x \in \mathit{out}(\Gamma)\} && (3)
\end{aligned}
$$

where t_x *is obtained by recursively applying the algorithm for* $\{\sigma \in L_\delta^\star \mid x \cdot \sigma \in \mathcal{F}\}$ *and* Γ'*, with* $\Gamma \xrightarrow{x} \Gamma'$.

In the algorithm, $\mathcal{F}$ is the set of traces after which *out-set* inclusion need to be checked. The first choice terminates the generation procedure to ensure test experiment stop at some point even though the specification may include infinite behaviour. The second choice gives a next input to the implementation. As inputs are always enabled, this step will never result in deadlock, thus no terminal state *pass* or *fail* can be reached. The third step checks the next output of the implementation. Any implementation producing an output *x* which does not belong to out(Γ) will result in a *fail* terminal, indicating it is not a conformance implementation.

This test generation algorithm guarantees to generate sound test cases with respect to *ioconf* and *ioco*, and the set of all possible test cases that can be obtained is complete. The proof can be found in [Tre96].

5.2 Application to Synchronous Circuits

Two examples are used to illustrate the approach of applying IOLTS-based formal conformance testing to validating synchronous circuit designs. One is a JK flip flop, the other is a single pulser which has already been specified in chapter 3.

5.2.1 DILL Specifications of the Examples

Recall that the DILL approach for specifying synchronous circuits is based on a clock cycle-by-cycle (see section 3.3.4). Clock signals just contribute to timing references and are not relevant to functionality. Thus they are often omitted in top level specifications. On each clock cycle, primary outputs and internal outputs are decided by primary inputs and internal inputs.

A JK flip-flop is a single-bit memory element with control inputs *J* and *K*. If they are both set to 0, the flip-flop stays in the same state. If they are both set to 1, the flip-flop inverts its current value. If *J* and *K* are set to different values, the value of *J* is stored. The output is conventionally called *Q*, while its complement is *NQ* (not *Q*). Unlike the specifications in chapter 3, the JK flip flop specification below fixes the order of inputs *J*, *K* and outputs *Q*, *NQ*. As discussed before, fixing orders might cause deadlock when components are connected. However, because testing just concerns the higher level behavioural specification, no connection is actually needed. By restricting the order of events, the state space can be substantially reduced when there exist multi-inputs and/or multi-outputs in a component. In the single pulser specification, implementations are allowed to assert the output pulse either on the positive going or negative going transitions of a input pulse, thus the specification is a non-deterministic one. The specification can be found in section 3.3.5.

```
behaviour JK [J, K, Q, NQ] (0)                              (* initial state is 0 *)
where
  process JK [J, K, Q, NQ] (dtQ : Bit) : noexit :=
    J ?newJ : Bit; K ?newK : Bit;                           (* get new J and K *)
    ( [(newJ eq 0) and (newK eq 0)] ->                      (* both 0 - same state *)
        Q !dtQ; NQ !not(dtQ);                               (* output current values *)
        JK [J, K, Q, NQ] (dtQ)
    []
      [(newJ eq 1) and (newK eq 1)] ->                      (* both 1 - flip state *)
        Q !not (dtQ); NQ !dtQ;                              (* invert outputs *)
```

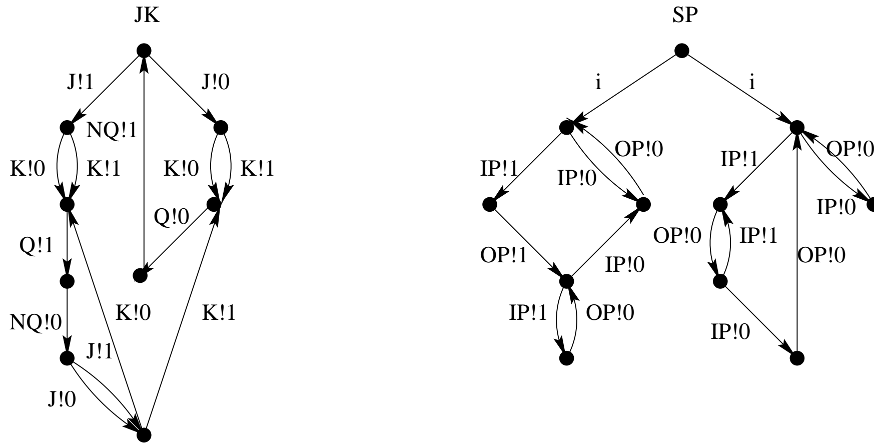


Figure 29: The LTS of the JK flip flop and Single-Pulser

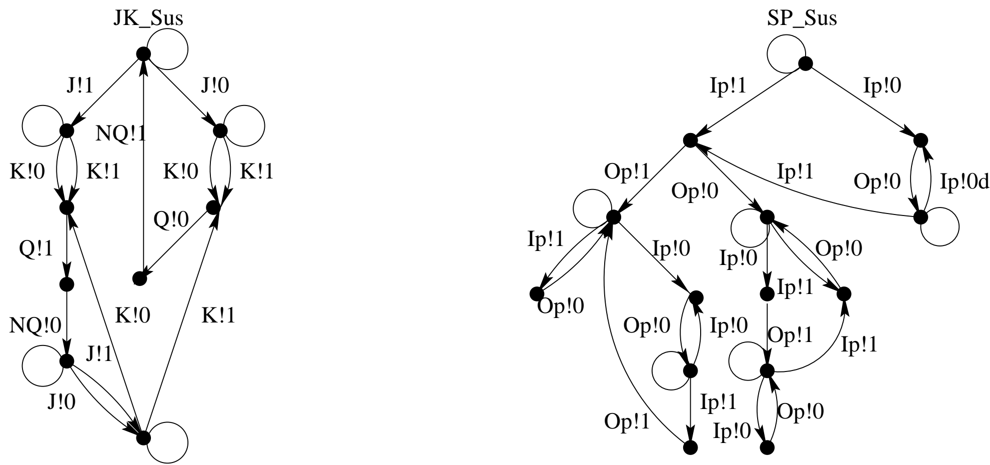


Figure 30: Suspension automata of JK flip flop and Single-Pulser

```
        JK [J, K, Q, NQ] (not (dtQ))
    []
      [newJ ne newK] ->                        (* both differ - take J *)
      Q !newJ; NQ !not (newJ);                 (* use J as input *)
      JK [J, K, Q, NQ] (newJ) )
endproc (* JK *)
```

5.2.2 LTSs, Suspension Automata and Test Cases

The LTSs that are observationally equivalent to the above LOTOS specifications appear in figure 29. Observational equivalence is used here since conformance testing relates only to external behaviour of circuits. The equivalence preserves all external behaviour and has a much smaller state space compared to that of the original specifications. Figure 30 shows suspension automata built from the LTSs. Self-loops in this figure denote δ (quiescent state) actions. If a specification is deterministic, such as the case of the JK flip flop, its suspension automaton is almost identical to the LTS except for the δ transitions. This is because suspension automata are obtained by determinising LTSs and adding necessary δ transitions. Figure 31 presents several possible tests generated from the automata using the algorithm explained in the preceding section.

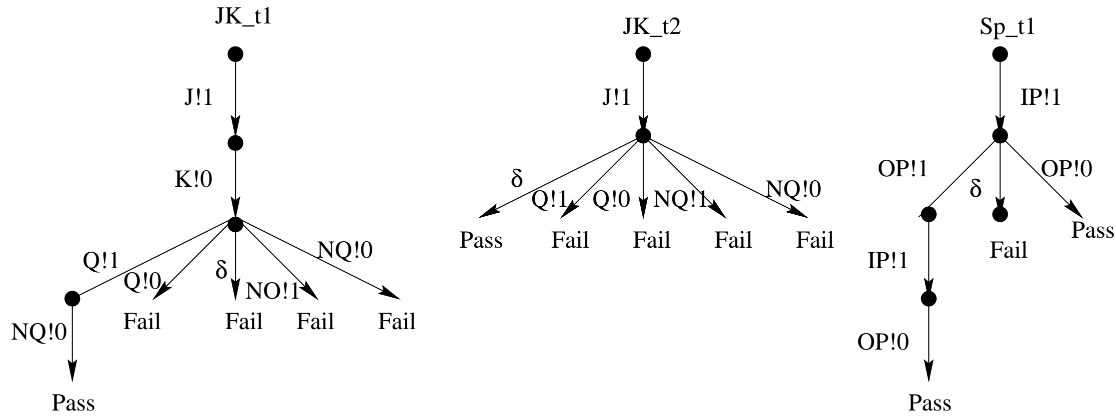


Figure 31: Several tests of JK flip flop and Single-Pulser

The modelling approach of DILL has some implications for testing. Firstly, LOTOS events represent stable signal values in a specific clock cycle. Therefore all the events in figure 31 are stable signals in a certain clock cycle. It follows that applying inputs and observing outputs according to the tests should also be conducted when a circuit is stable. This is not a problem for synchronous circuits since clock cycles are chosen such that circuits have enough time to settle down. Secondly, as it is assumed the clock cycles are slow enough, stable values of inputs and outputs are guaranteed to appear once in every clock cycle, so there is no need to worry about δ actions which indicate the absence of outputs. This is why the weaker relation *ioconf* is used for testing synchronous circuits. For the same reason it is also less interesting to generate tests cases similar to *JK_t2* that check absence of outputs. They are therefore excluded from the test generator. Finally, as discussed earlier the order of inputs and outputs is fixed to restrict the state space. Test case *JK_t1* gives a *Fail* verdict when the first *NQ !0* is observed. This would not have happened if the full state space had been generated, so it is a fake failure state. The way to solve this problem is discussed in the following section.

These two examples also indicate why the *ioconf* relation is a suitable implementation relation for validating synchronous circuits. If a specification is deterministic then *ioconf* requires that, in each clock cycle, for all possible input sequences, all the outputs of an implementation agree with those given by the specification. This is strong enough to distinguish erroneous implementations from correct ones. On the other hand, it also permits non-deterministic specifications to be tested. Because every output will appear once in a clock cycle, a non-deterministic specification will have one or more outputs having contradictory output values, i.e. the output may produce either *1* or *0* in that clock cycle, as in the case of the Single Pulser. This can be properly captured by the *ioconf* relation. For example if the input is initially *0*, after it changes to *1* the output of a positive edge implementation should be *1*, or *0* for a negative edge implementation. As seen in test case *Sp_t1* of Figure 31, both design decisions can pass the test so implementation freedom is respected.

5.2.3 Test Generation and Execution

The test cases generated from the algorithm in section 5.1.6 have the form of trees. This might have a straightforward mapping to TTCN (Tree and Tabular Combined Notation [ISO91]), a standard form of test suites. However, it is found that testbenches, which aim to automatically provide test cases to HDL simulators and report test verdicts to testers, cannot be easily developed from test trees. In addition, the coverage of a test suite is not easily measured if it is expressed in test trees. This thesis hence uses *test traces* instead of *test trees*.

Recording test cases as input output sequences is a very common way in engineering practice of testing digital circuits. For example, test case *JK_t1* can be stored in a file of the form: *J!1; K!0; Q!1; NQ!0; Pass*, indicating that when inputs are *J=1, K=0* the outputs should be *Q=1, NQ=0*. All the other branches which lead to *fail* states in this test tree are in fact not necessary: as the digital signals are assumed to be strict

binary, if *Q* is not *1*, then it must be *0*. So when *0* is generated from the implementation, the *fail* verdict can be obtained automatically by comparing it with the test trace; δ can be omitted as discussed. The other branches are fake tests, which are the consequence of fixing the order of inputs and outputs. It is concluded that test trees can be transformed to a test trace in which all transitions leading to the *Fail* verdict in the tree are not explicitly recorded. When implementations have outputs different from the one defined in a test trace, a *Fail* verdict should be generated automatically.

This method works well with deterministic specifications. However when the specification has non-deterministic behaviour, simply generating traces from test trees raises problems. For example, the test tree of *Sp_t1* cannot be rewritten as *Ip!1; Op!1; Ip!1; Op!0; Pass* and *Ip!1; Op!0; Pass*. If a positive triggered implementation were tested by the first case, it would be given a *Fail* verdict. Conversely, a negative triggered implementation would fail the second test. Actually, both of them might be correct implementations. The problem is that an implementation has to pass all the test cases in a test suite before it is regarded as correct. But for this example, only passing one of the test cases is necessary. This is solved by marking outputs at a contradictory branch, i.e. the branch where the same output may produce either *1* or *0*. Failing a marked output in a test run gives an inconclusive verdict, indicating that the IUT is allowed to produce an output other than the one dictated by the test.

At some node of a suspension automaton, suppose the test generation program finds that there are two possible output transitions with the same gate offering different values. Both of the outputs should be marked when the corresponding sequences are generated, meaning they are not necessarily matched by an implementation. Coming back to the example above, the tests then become *Ip!1; Op!1⋆; Ip!1; Op!0; Pass* and *Ip!1; Op!0⋆, Pass*. When output *Op!1* from an implementation is compared to the second test case, the ⋆ means this output does not have to be matched. Another test trace is then applied. In this case, outputs are matched so testing continues to analyse if the subsequent behaviour is satisfied.

To get the test traces, generation is mainly based on traversing suspension automata. Referring to Tretmans' algorithm in section 5.1.6, if Choice 1 is made, generating a test case is complete. Appending an input action to a trace corresponds to selecting Choice 2 in the test generation algorithm. Appending an output event, possibly with a ⋆ mark, equates to selecting Choice 3.

As specifications usually have infinite behaviour, especially if they involve iterations, a test case can hardly be a complete trace unless the circuit has a deadlock state. Therefore a test suite can never cover all the behaviour of a specification. How to generate a test suite with good coverage is an important but hard theme for the testing community.

If covering all behaviour is not achievable, then covering all transitions might be a second-best choice. A suspension automaton is a directed graph. Generating a sequence that visits every edge in a graph at least once is the **Chinese postman problem** [EJ72]; the generated sequence is termed a **transition tour**. A single transition tour exists only for a strongly connected graph, i.e. the graph in which every node has a path to every other node. Otherwise, more than one tour is needed to cover all the edges. As suspension automata may not be strongly connected, it is not possible to make direct use of transition tour generation algorithms (e.g. [Hol91]), which guarantee the shortest tour for strongly connected graphs. In the work presented here, the approach suggested in [HYHD95] is adopted because it is suitable for all kinds of directed graphs. In this method, depth-first search (DFS) is used whenever possible as it naturally records the transitions traversed. When an un-visited edge cannot be reached by DFS, breadth-first search (BFS) is exploited to find a state that has an unvisited edge; DFS then continues from this state. The whole procedure repeats until there is no unvisited edge in the graph.

The CADP toolset supports an application programming interface that allows user-written programs to manipulate the state space of a given LOTOS specification. This interface is exploited to program the test generation algorithm based on transition tour. The algorithm is given below. Note that test cases are influenced by the order in which the edges of a suspension automaton are stored. This order is adjustable by changing parameters passed to CADP. If more coverage is required, the test generator can be re-run by using different parameters.

Test generation based on transition tour:

```
TestGen()
{                                   /* First produce suspension automaton from the LTS */
  InitStat = SusAutGen();                    /* InitStat is the initial state of SusAut */
```

cycle	J	K	Q	NQ	cycle	J	K	Q	NQ
cycle1	1	1	1	0	cycle5	0	0	1	0
cycle2	1	1	0	1	cycle6	1	1	0	1
cycle3	0	1	0	1	cycle7	0	0	0	1
cycle4	1	0	1	0	pass				

Table 2: Test suite for JK flip flop

```
state = InitStat;
while (1) {
 while (1) {
                    /* DFS traverse from the state, until no unvisted edge can be found */
                    /* at the same time, for each output transition traversed, */
                    /* mark it with '*'  if it has contradictory neighbour */
 state1 = DFS(state);

                    /* When DFS cannot find a state with untraversed edges,*/
                    /* do BFS search from state1 to look for the state */
                    /* that has an untraversed edge, mark the edges at the same time */
 state2 = BFS(state1);
 if (state2 != NULL) {                                   /* find a state */
   ShortestPath(state1, state2);
   AppendPathtoTrace();
   state = state2;}
 else {                                                  /* not find an untraversed edge */
   PrintTraceMark();
   break;}}

state1 = Initstate;
state2 = BFS (state1);
if (state2 != NULL) {
  ShortestPath(state1, state2);
  AppendPathtoTrace();
  state = state2;}
else return}}
```

Table 2 shows a test case for the JK Flip-Flop generated from the implemented program. It only requires 7 clock cycles to test the flip flop, and the test covers many important behaviours. For example, unlike other input combinations, inputs *J=1, K=1* do not produce unique outputs. The test case thus uses several clock cycles to test this feature. Table 3 shows the test suite for the Signal Pulser. The first test is actually for the negative triggered implementation, and the second is for the positive triggered implementation. Any correct implementation will pass one of them, and will have an inconclusive verdict when tested by the other. Thus those getting the *fail* verdicts are really incorrect implementations.

Each tour generated in this way is a test case and is saved in a test file. The accumulated test cases are passed to a VHDL simulator that simulates the lower level implementation of a circuit. A VHDL testbench is designed to allow the test cases to be applied and executed against the VHDL description. The testbench is in fact a VHDL program which consists of two processes that are executed concurrently. The first process generates clock signals for the circuit under test. The second process reads the test suite file and generates signal stimuli according to the inputs of each test case. It also compares the outputs generated by the VHDL simulator with the output values required by the test case, giving a *Fail* or inconclusive verdict and aborting the simulation if they are not the same. The testbench also has to determine when to apply the input stimuli and to check the output result. This needs some knowledge of the circuit realisation, such as

cycle	Ip	Op	cycle	Ip	Op	cycle	Ip	Op
cycle1	0	0	cycle4	0	1	cycle7	1	0
cycle2	0	0	cycle5	0	0	pass		
cycle3	1	0⋆	cycle6	1	0			
cycle	**Ip**	**Op**	**cycle**	**Ip**	**Op**	**cycle**	**Ip**	**Op**
cycle1	1	1⋆	cycle3	0	0	cycle5	1	0
cycle2	0	0	cycle4	1	1	pass		

Table 3: Two test cases for Single Pulser

the propagation delays of components in the circuit. Special care should be given to those outputs which are marked with ⋆. Between two test cases, a reset signal is generated by the testbench to re-initialise the circuit under test. The assumption is made that a circuit can always be correctly reset. The LOTOS specifications discussed previously do not specify reset behaviour, so a test need not be generated to ensure that reset is correctly achieved.

5.2.4 Further Discussion

When a suspension automaton is strongly connected, the transition tour algorithm in section 5.2.3 generates a single test case, such as for JK flip flop. Otherwise, the number of test cases is the number of the strongly connected sub-graphs in the suspension automaton, such as in the case of the Single Pulser.

The specification of Single Pulser is non-deterministic in that it allows two kinds of implementations. But the behaviour of each kind is actually deterministic. Each strongly connected sub-graph in the suspension automaton corresponds to an implementation, and a test case is generated for it. If an implementation is tested by a test case that is not for its kind, an inconclusive verdict arises telling the tester that the test case applied is not a proper one. The test suite has the property that there is always a test case which can characterise implementations.

However, if the behaviour of an implementation is non-deterministic, for example, if the single pulser is allowed to assert its output pulse at either negative edge or positive edge transitions of its input, the suspension automaton becomes a strongly connected graph. In this case, there is only a test case generated by the algorithm. Many of the implementations, no matter if they are correct or not, will get the inconclusive verdicts from this test case because when they assert output on the positive edge transition, for example, they may meet a transition in the test case requiring the output on negative edge. Hence for the specifications which allow non-deterministic implementations, the algorithm is not so efficient due to the frequent inconclusive verdicts. In the next section, a solution is proposed for non-deterministic implementations.

In fact, non-deterministic digital circuits are really rare. Normally people expect digital devices to have predictable responses to all their inputs. Therefore, the test traces generated are satisfactory in most cases.

5.3 Application to Asynchronous Circuits

Apart from using *ioco* instead of the *ioconf* relation in the test generation algorithm, the approach of applying conformance testing to validating asynchronous circuit designs virtually has no difference from that for validating synchronous circuit designs. Following the way of the previous section, two examples of asynchronous design are used to facilitate the explanation.

In chapter 3, an asynchronous first-in-first-out buffer is specified. Designed for dual-rail data paths, this buffer has two input *InT, InF* and two output *OutT, OutF*. It is assumed to be empty initially. When *1* appears on *InT* or *OutT*, the datum on the datapath is *1*. When *1* appears on *InF* or *OutT*, the data is *0*. Lines should be reset to *0* between two transformations. The specification will not be repeated here as it can be found in section 3.4.7. Figure 32 gives its LTS (minimised with respect to observational equivalence), suspension automaton of the LTS, and several tests. As seen, because the LTS is deterministic the automaton has almost the same structure except for the δ transitions, which are represented as circles in

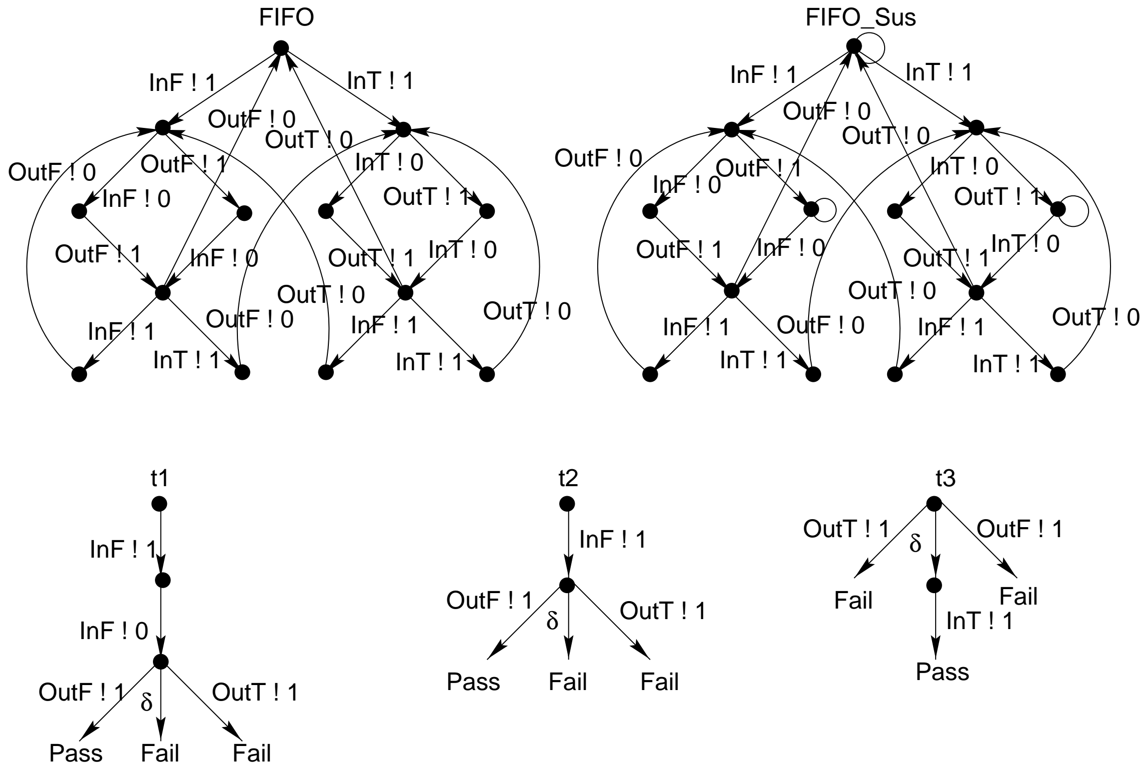


Figure 32: LTS, suspension automaton and several tests of FIFO

the figure. Test *t1* provides two inputs then checks the output of an implementation. If output *OutF* changes, the implementation passes the test. However if *OutT* changes or if there is no output, the implementation fails the test. Similarly, test *t2* checks output after one input is provided. Test *t3* checks output right away. Output changes from the initial states are erroneous behaviour so testing should stop after they are observed. Only after the δ transition, meaning that no output is produced, can testing continue. Note that this test has a δ transition that does not lead to a terminal state, which could never happen in the test cases for synchronous circuits.

The second example is a selector, which has also been introduced in section 3.4.4. This is a specification which allows non-deterministic behaviour in implementations: after an input change on input *Ip*, depending on implementations, either *Op1* or *Op2* may change. Figure 33 gives its LTS (minimized with respect to observational equivalence), suspension automaton of the LTS, and one of the test cases. *Selector-test* indicates that after the input *Ip ! 1*, the implementations producing either *Op1 ! 1* or *Op2 ! 1* will pass the test, which respects the implementation freedom required by the specification.

The above two examples illustrate that the *ioco* relation is suitable for testing asynchronous circuit designs. On the one hand it is strict enough to reject erroneous designs, and on the other hand it supports implementation freedom by passing all possible correct implementations.

Specifications of asynchronous circuits sometimes permit some of their outputs to be produced in any order.[4] This is usually modelled as interleaving of these outputs in DILL. The situation is relatively common in asynchronous circuit specifications. As in the case of non-deterministic behaviour, their suspension automata also contain nodes which have more than one outgoing transition labelled with output actions. This is not a coincidence because interleaving outputs actually introduces non-determinism. Concrete implementations usually produces these outputs in a fixed order.

In the previous section, the problem caused by non-determinism was solved by marking output transitions which have contradictory neighbours in suspension automata. This technique can be easily extended here for asynchronous circuits: each output transition which has other neighbouring output transitions is

[4]In synchronous circuits, the order of outputs is artificially fixed because it does not influence the functionality.

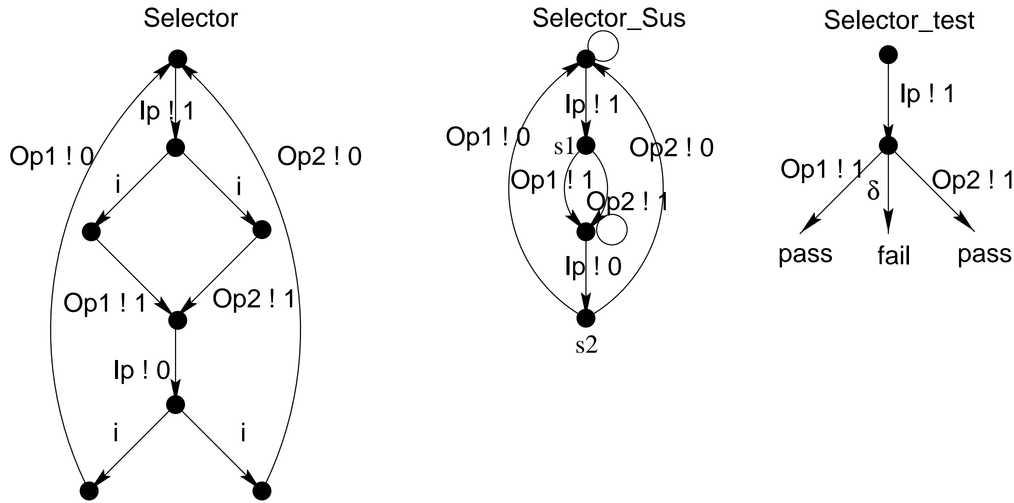
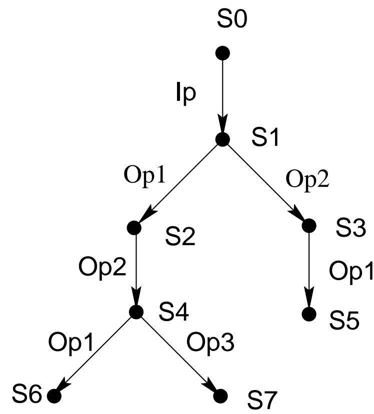


Figure 33: LTS, suspension automaton and one test of Selector



A possible test trace:

Ip, Op1(*s1), Op2, Op1(*s4), ...

Op3(*s4), Ip, Op2(*s1), Op1...

Figure 34: Nodes with more than one outputs and its test trace

marked during test generation, indicating that this output may not be matched by implementations under test since other outputs are allowed to be produced. As discussed in section 5.2.4, this method is not so efficient when the behaviour of an implementation is non-deterministic. The problem is that when an inconclusive verdict is reached, a test run is aborted and other test cases (if any) should applied. However, the test case should be still valuable if other neighbouring outputs can be found so that the test run can continue. To achieve this, all marks are extended with the source nodes of the outputs, i.e. output transitions are marked with $\star$ as well as their source nodes when they have neighbouring outputs. Obviously outputs with same marks in a test suite are neighbours of each other in corresponding suspension automaton. In this way the branch structure of a tree is mimicked by a trace. Since the transition tour algorithm is able to cover all the transitions in a suspension automaton, if an implementation cannot agree with all the outputs with a certain mark, *fail* verdict should be issued. This technique requires a testbench which is able to search the whole test suite for marks.

Figure 34 is an example for this revised algorithm. If an implementation has the behaviour *Ip, Op1, Op2, Op3,* $\cdots$, it will follow *Ip, Op1, Op2* in a test run, but when the output *Op3* fails at *Op1(*$\star$*s4)*, a testbench should look for another output with the same mark to see if the two can match. In this case it finds *Op3(*$\star$*s4)*, then the testing continues. If an implementation behaves as *Ip, Op3,* $\cdots$, there will be no output marked with *(*$\star$*s1)* that can match the *Op3*; the implementation is therefore regarded as erroneous.

As far as the testbench is concerned, it will be more complicated than its synchronous counterpart. To

1	2	3	4	5	6	7
InF=1	InF=0	OutF=1	InF=1	OutF=0	OutF=1	InF=0
8	9	10	11	12	13	14
InT=1	OutF=0	InT=0	OutT=1	InT=1	OutT=0	OutF=1
15	16	17	18	19	20	21
δ	InF=0	OutF=0	InT=1	OutT=1	InT=0	InT=1
22	23	24	25	26	27	28
OutT=0	OutT=1	δ	InT=0	OutT=0	δ	pass

Table 4: Test suite of FIFO

1	2	3	4	5	6	7
IP=1	Op1=1$\star$s1	Ip=0	Op1=0$\star$s2	δ	Ip= 1	Op2=1$\star$s1
8	9	10	11			
δ	Ip=0	Op2=0$\star$s2	pass			

Table 5: Test suite of Selector

be able to deal with non-deterministic implementations, it should be able to search a whole test suite for marks. Normally a search should be carried out in the rest of a trace when an inconclusive point is met, so that testing can go forward. However, sometimes such marks only exist in the previous part of the trace, forcing the search to go backward. This means that there may exist loops during testing. The testbench thus should have a strategy to break such a loop. A testbench also needs to maintain a timer. In the real world, no component really has unbounded delay, so when a δ transition is seen, the testbench uses the timer to record the time that elapses. If there is no output within a certain amount of time, the δ transition is assumed to be satisfied, otherwise the *fail* verdict will be given. The value of the timer relies on the delays in a circuit. A testbench will also have to decide when to provide inputs. For the test case *t1* in figure 32, if *InF ! 0* is provided too late after the first input *InF ! 1*, an output may have already been produced. The behaviour should be tested by other test cases such as *t2*. But as testers may not aware this, *t1* may still be used, which will produce faulty test results.

As a conclusion of this section, the test suites of the above two examples produced by the revised transition tour algorithm are given in table 4 and 5. Both have just one test case. The one for FIFO has a length of 28 transitions, with a length of 11 transitions for the selector. The second test suite is a test with inconclusive marks. A selector which insists on sending its input to *Op1* can follows the test sequence *1, 2, 3, 4, 5, 6, 2, 3,* $\cdots$, a loop that a testbenches must break.

5.4 Case Study

This section evaluates the approach by generating test cases for a DILL specification of a circuit, then executing them against its implementation described by VHDL code. This is a synchronous circuit: the BlackJack Dealer[SK96], a famous card game which is also called pontoon or “21”.

A BlackJack Dealer is a device which plays the dealer's hand of a card game. The inputs of the circuit are *Card_Ready* and *Card_Value* (Ace..King, Clubs..Spades). Its outputs have boolean values: *Hit* (card needed), *Stand* (stay with current cards) and *Broke* (total exceeds 21). The *Card_Ready* and *Hit* signals are used for a handshake with a human operator. Aces have value 1 or 11 at the choice of the player. Numbered cards have values from 2 to 10. Jack, Queen and King count as 10. The Black-Jack dealer is repeatedly presented with cards. It must assert *Stand* (when its score is 17 to 21) or *Broke* (when its score exceeds 21). In either case the next card starts a new game. Figures 35, 36 and 37 are the implementation of the circuit given in [SK96].

In the DILL specification of the BlackJack dealer, a new data type *Value* is defined to represent the

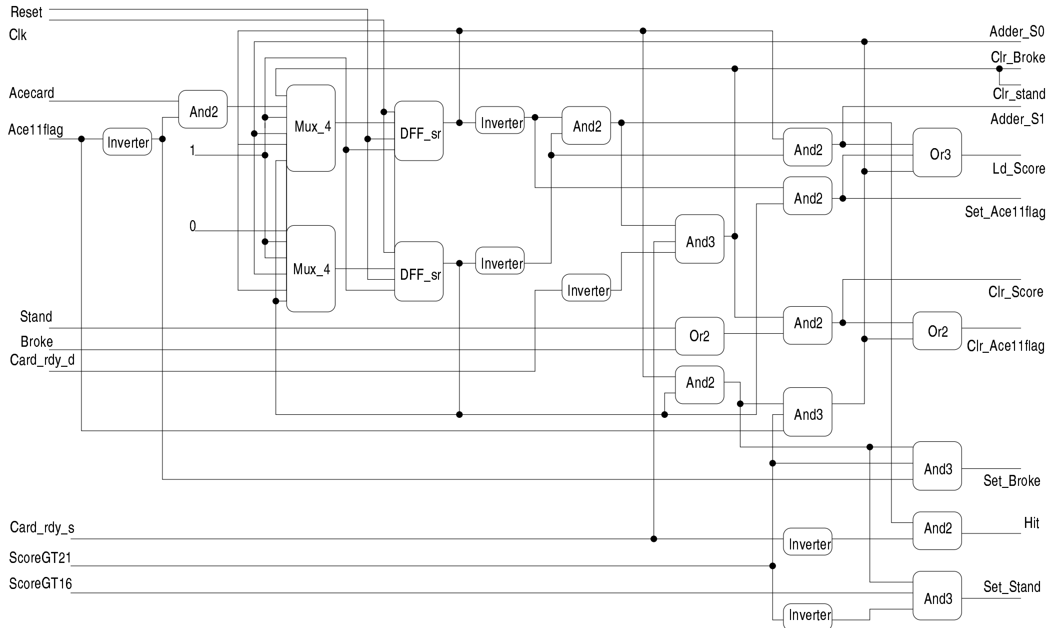


Figure 35: The controller of the Black-Jack Dealer

card value. Although the LOTOS standard data type *NaturalNumber* might appear suitable, CADP cannot generate the corresponding LTS for an infinite data type like this. The key point in the specification is how to handle the ambiguous value of an Ace. To solve the problem, the specification uses the method given by [WP80]. Specification behaviour occupies about 80 lines including comments.

Using CADP and the test generator program implemented for the algorithm in section 5.2.3, a test suite for the Black-Jack Dealer was derived. The test suite is able to test 181 different hands of cards that a dealer may hold. The VHDL implementation given in [WP80] was evaluated against this test suite.

Although the circuit was expected to pass the test suite, a *Fail* verdict was recorded after the dealer was given the following cards: 5, 5, 3, 2, 1, 10. In this case the dealer should be *Broke* because the sum of the cards is 26, which exceeds 21. However the circuit outputs neither *Stand* nor *Broke* since it considers the total to be just 16. Other card combinations including an Ace that should cause *Broke* exhibited the same problem. This indicated that the problem was related to processing an Ace.

The circuit should initially take an Ace as 11. It should be re-valued as 1 (subtracting 10 from the sum) the first time the result would be *Broke*. If the following cards would make the sum exceed 21, no re-valuation should be done as no Ace is 11. By carefully simulating the traces which led to the failure, it was discovered that the given benchmark design still re-values the Ace card, so the circuit is not *Broke* in this case. In the design of the BlackJack Dealer, there is a flag register (*Ace11Flag* in [SK96]) indicating if there has been an Ace evaluated as 11. The problem of the circuit is that this register is not reset to zero properly after an Ace is reset to 1, because the effective duration of the signal used to reset it is too short.[5] By slightly modifying the circuit to remove the cause of this short duration, the circuit was able to successfully pass the test suite.

5.5 Related Work

For validating hardware designs, simulation has been and is still the predominant method in industry. Test cases for simulation are mainly manually defined or randomly generated. Recent developments for solving the problem lie in combining formal methods with traditional simulation techniques. In [VK95], tests are

[5] One of the registers in the design of the circuit is negative effective, but all the other registers are positive effective. Consequently the effective duration of *ClearAce11Flag* is just half a cycle, which is not enough to clear the *Ace11Flag* signal. The circuit designer might wish to save one clock cycle to improve the speed of the circuit by using a negative triggered register.

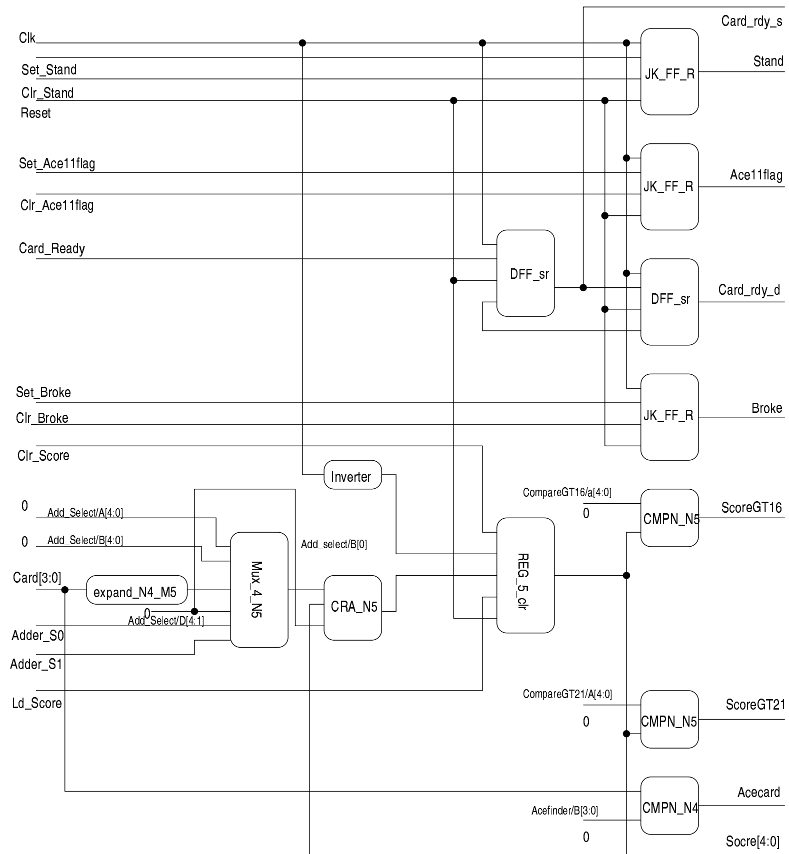


Figure 36: The datapath of the Black-Jack Dealer

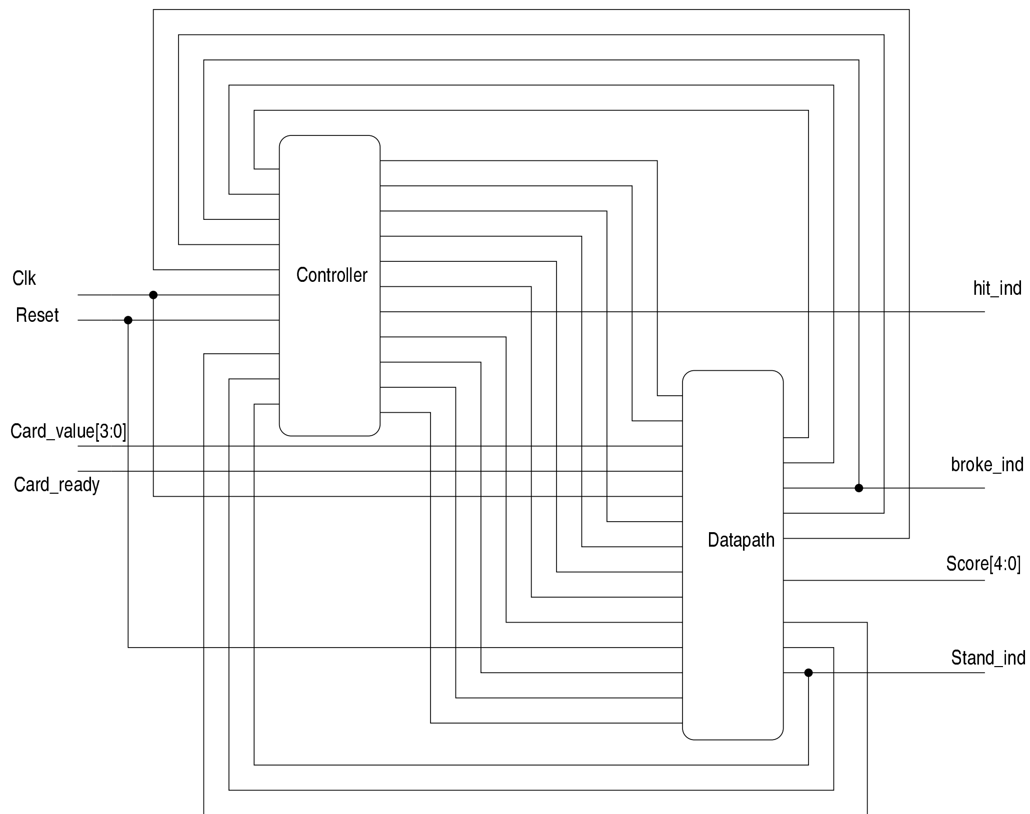


Figure 37: Control-Datapath communication

generated from behavioural VHDL programs using traditional software testing techniques. In [HYHD95, MAH98], test generation is based on an FSM (Finite State Machine) or an ECFM (Extracted Control Flow Machine), which represents the control logic of a circuit. The generated test cases are then applied to both higher level and lower level specifications in Verilog or VHDL, verdicts are obtained by comparing outputs from two levels. The basic idea of these two papers is quite similar to the one presented here, except that they extract a formal model from circuit design and use the techniques essentially based on FSM testing theory. But in this thesis, tests are derived from higher level specifications using conformance testing theory for LTSs. In [RSM97], test generation does come from a higher level specification of an FSM, then applied to a VHDL simulator, but it cannot handle the case where specifications involve nondeterminism. The aim of that paper is to fill the gap between the abstract tests and concrete signals; test generation is based on a commercial tool.

Finally, within the CADP toolset a test generation tool TGV [FJJV96] is under development, the implementation relation exploited is very similar to the *ioconf* used in this paper. TGV had not been released by the time this thesis was finished, thus a comparison could not be given.

5.6 Conclusion

In this chapter, the framework of formal methods in protocol testing was used for testing digital circuits. The chapter first gave a brief introduction to formal conformance testing based on the formalism of LTSs, then focused on a recent extension to this theory, namely testing implementations which are modelled as *IOLTSs*. It is believed that an *IOLTS* is a more faithful model of digital hardware than an LTS. Subsequently two implementations relation *ioconf* and *ioco* and associated test generation algorithm were presented. From the examples and the case study, it can be seen that this formal framework can be successfully applied to testing digital circuits designs

A tool *TestGen* has been implemented in a C programm which produces suspension automata from DILL specifications and generates test suites based on the transition tour of the automata. The main purpose of developing such a program is to automatically generate test suites which have reasonable coverage, and to facilitate automatic test execution. To achieve this, a testbench written in VHDL was developed to bridge the test cases and VHDL simulator. A revised version of the generation algorithm was also implemented, which allows non-deterministic implementations to be tested. This revised algorithm requires a relatively complicated testbench, which has not been implemented at the current stage.

The case study of the Black-Jack dealer shows the benefits of the approach. By executing test cases on a more detailed model of digital circuits, here it is a VHDL description which contains timing characteristics of components, it is possible to reveal subtle bugs which cannot be captured by analysing a formal model. The problem identified in the case study actually related to the timing characteristics of the circuit. Although a DILL specification does not contain timing information at all, timing bugs can still be discovered by the approach.

6 Specification and Analysis of Timed Circuits

This chapter specifies circuit behaviour by explicitly including quantitative timing magnitudes. In the sequel, such specifications are termed *timed specifications* and the specified circuits are *timed circuits*. In the background section, the specification language ET-LOTOS(Enhanced Timed-LOTOS [LL97]) is introduced. It is followed by the investigation of building the DILL model of timed circuits. A timed circuit has three parts: functionality, delays and timing constraints. As will be discussed, the model is compositional and the untimed DILL specifications are just the special cases of the timed ones. The chapter also gives the specifications of various delays and timing constraints. For analysing timed circuits, TE-LOLA (Time Extended LOTOS Laboratory [PLR95]) is used. Finally a case study is investigated to examine the approach.

6.1 Background

So far in the thesis, digital circuits specified are untimed, i.e. quantitative timing characteristics are abstracted away in specifications. For synchronous circuits, clock cycles are assumed to be slow enough so that the period of a clock can be abstracted as a time unit. For asynchronous circuits, only those designs based on unbounded delay models are modelled.

However, timing characteristics have never been something negligible in digital circuit design. Timing analysis is critical because it determines if a circuit can function correctly. For example, an improper clock cycle is a disaster for synchronous circuit. In asynchronous circuits, potential race conditions and hazards, which are caused by propagation delays of components, can result in a circuit malfunctioning. In addition high-speed performance is a very important criterion in today's competitive market. Formalisms which support timed specification and analysis are therefore highly desirable.

6.1.1 ET-LOTOS in Brief

The thesis chooses ET-LOTOS as the formalism for specifying timed circuits. ET-LOTOS is closely related to the future ISO standard E-LOTOS (Enhancements to LOTOS [ISO98]). It is hoped that the work presented here will be easily transferable to E-LOTOS once the standard is mature.

ET-LOTOS supports both discrete and dense time domains. Informally, in a discrete domain time progresses in discrete steps. In a dense domain however, it is always possible to find a time value between any two given time values. The discrete time domain is represented by the natural numbers, and the dense time domain by real or rational numbers. In ET-LOTOS, only countable time domains (such as rational numbers) are permitted in order to give operational semantics using Labelled Transition Systems. Time domains are defined as data types. This makes ET-LOTOS very flexible as time values can be treated like any other data values.

Three new operators relevant to time are introduced, namely delay, life reducer and time measurement.

Life Reducer: Action-prefix is extended in ET-LOTOS: the expression $g\{d\}$ means g will not be offered after d. In other words, g can only occur in the interval of time *[0, d]*. The temporal attribute $\{d\}$ is termed a life reducer.

The precise semantics of $g\{d\}$ is as follows: if after a delay time d, the behaviour $g\{d\}; \ldots$ has not been performed g, the g offer is removed without executing the subsequent behaviour, i.e. the process starts behaving like the idle process **stop**. Note that the life reducer does not enforce the execution of g within the interval *[0, d]*, it just states that g cannot occur outside this interval. When there is no life reducer, the standard LOTOS syntax applies. The default value of the life reducer is thus ∞ for observable actions, which matches the LOTOS semantics that observable events can happen at any time.

Comparatively, applying the life reducer to the internal event, **i** $\{d\}$, means that **i** *must* occur non-deterministically within the next d time units. Necessity and non-determinism apply because internal actions are not controlled by the environment; in particular, the time of occurrence is decided by system itself. Nonetheless, an alternative action may pre-empt the occurrence of an internal action. If the life reducer is omitted, it is regarded as **i** $\{0\}$, i.e. the internal event must occur at once (if at all).

Delay: The delay operator Δ^d means that the subsequent behaviour will be delayed by *d*. In ET-LOTOS a time value is relative to the instant when the previous action occurs. So the behaviour *a;* $\Delta^d P$ will delay for *d* after event *a* occurs and then behave like *P*.

Time Measurement: The time measurement operator @*t* is used to measure the time elapsed between the instant when the event has been offered and the instant when it occurs. The time value is stored in *t*. In ET-LOTOS, time measurement can be used for both observable actions and internal actions. For observable actions, the time measurement variable *t* can appear in selection predicates. For example *a* @*t* $[t \leq 5]$*; P* denotes a behaviour which can perform *a* only within the first 5 time units and then behave like *P*. The time when it takes place is recorded in *t*.

Apart from these basic operators, there are also some shorthand notations for flexibility and convenience:

Generalized Life Reducer on Observable Action: *g* @*t* $[d1 \leq t \leq d2]$*; P* can be rewritten as *g* $\{d1,d2\}$*; P* provided that *t* does not appear in process *P*. It can also be expressed in terms of the delay operator and the life reducer, such as Δ^{d1} *g* $\{d2\text{-}d1\}$*; P*; the same condition applies.

Generalized Life Reducer on Internal Action: The behaviour Δ^{d1} **i** @*t* $\{d2\}$*;* *[t+d1/t]P* can be rewritten as **i** @t $\{d1,d1+d2\}$*; P*, where *t+d1/t* means every *t* appearing in process *P* is replaced by *t+d1*.

The formal semantics of ET-LOTOS is given by labelled transition systems. There are two kinds of transitions: discrete and timed. Discrete transitions correspond to the execution of actions. If *a* is an action, $P \xrightarrow{a} P'$ means that *P* may perform action *a* and then behave like P'. Timed transitions correspond to the passage of time. If *d* is a variable of sort *Time*, then $P \xrightarrow{d} P'$ means that *P* may idle for *d* then behave like P'. The semantics will not be discussed in detail here but two points are emphasized below. Full definitions of the semantics can be found in [LL94].

ET-LOTOS adopts maximal progress [Wan91] for hidden actions. Maximal progress means that if a hidden action can occur, it must happen now (unless an alternative action occurs) and should not be postponed. In other words, hidden actions are urgent in ET-LOTOS. In the DILL approach, each digital component is modelled as a process which usually synchronises with others. Input or output ports are modelled by LOTOS events. Ports used inside a design are hidden and their events become urgent under the assumption of maximal progress.

For **i** events, urgency is not always available. In the behaviour **i** $\{d\}$*;* **stop** the internal action can be postponed until *d* time units. But after that, it must happen (unless an alternative action occurs). An internal event is thus urgent only at its upper time bound.

6.2 LOTOS Model of Timed Circuits

Before developing a model to specify timed digital components, it is necessary to identify which kinds of timing characteristics need to be specified for digital designs. By intuition, timing characteristics are temporal relationships among inputs, among outputs, and among inputs and outputs. The relationship from input to output is normally called *delay*. It is the time interval between a signal change on an input and the resulting signal change on an output. The relationship among inputs is called a *timing constraint* in this thesis, meaning that digital circuits can work correctly only when the constraints are met. There is no need to specify the relationships among outputs directly, as they are determined by delays and timing constraints.

Several possible approaches exist to specify a timed digital component, classified here as either an *integrated method* or a *combined method*. In an integrated method, a digital component is specified in one process that deals with both functionality and timing. Although the integrated method may result in compact specifications, it is not a `structural' method and is hard to apply. The approach is not compositional in the sense that functional and temporal characteristics of a component are not merely combined. It is also important to have untimed behaviour as a simple case of timed behaviour, i.e. to be able to isolate pure functionality.

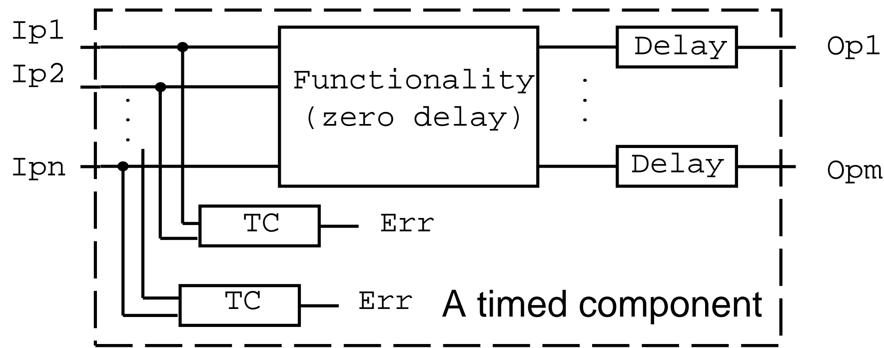


Figure 38: The specification model for a timed component

Attention has therefore been focused on developing combined methods. The idea is to separate the functionality and the timing characteristics into different processes, and then to combine them in an appropriate way.

The model adopted is a result of considerable experimentation with different approaches. The selected approach is called the *parallel-serial* model. As shown in figure 38, the functionality is assumed to be specified with no (in other words, zero) delay. Timing constraints (TC) are placed in parallel with the functional specification to check if input requirements are met. Delays are placed in series with the functionality to provide delay for each output.

Note that the *Err(or)* gates in the figure are for analysis purposes only; they have no counterpart in real physical components. It indicates that a timing constraint has not been met. It is found that modelling circuit behaviour under unexpected inputs conditions is really difficult, and even impossible sometimes. In fact, it is more important to detect and correct design errors than to know what happens after the errors.

If the timing constraints are void and the delays may be arbitrarily large, the timed model is equivalent to an untimed model. The model thus has the nice property that an untimed specification is a special case of timed one.

6.3 Specifying Functionality

The functionality part of a timed component has zero delay. In other words, outputs change immediately after an input change. Specifications of functionality are based on the model developed in section 3.2, i.e. the first model for basic logic gates. Only a small modification is made to reflect the zero delay condition, that is every output has a *0* life reducer. As discussed, the first basic logic model is very faithful to real world components except for its inertial delay assumption. Because no delay is associated with the functionality part of timed circuits, this shortcoming is therefore got rid of. Again, the specification of a *Nand2* gate is taken as an illustration.

```
process Nand2 [Ip1, Ip2, Op] : (dtIp1, dtIp2, dtOp : Bit) noexit :=
  Ip1 ? newdtIp1 : Bit [newdtIp1 ne dtIp1];                  (* one input is changed*)
  Nand2 [Ip1, Ip2, Op] (newdtIp1, dtIp2, dtOp)                (* repeat behaviour *)
[]
  Ip2 ?newdtIp2 : Bit [newdtIp2 ne dtIp2]                     (* other input is changed *)
  Nand2 [Ip1, Ip2, Op] (dtIp1, newdtIp2, dtOp]                (* repeat behaviour *)
[]
  let newdtOp : Bit = Apply (Nand, dtIp1, dtIp2) in           (* new Output *)
  Op ! newdtOp {0} [newdtOp ne dtOp] ;                        (* output change immediately *)
  Nand2 [Ip1, Ip2, Op] (dtIp1, dtIp2, newdtOp)                (* repeat behaviour *)
endproc (* Nand2 *)
```

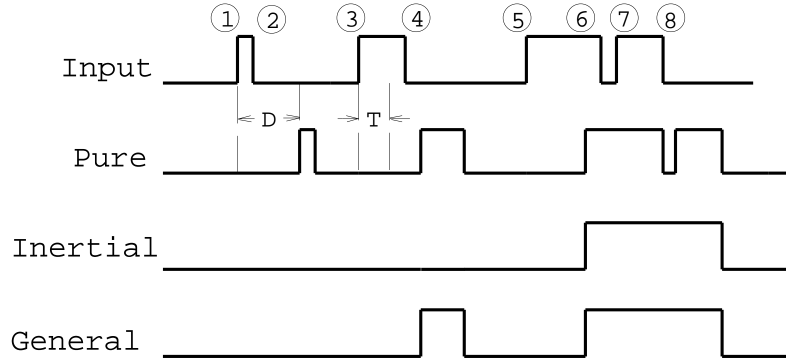


Figure 39: Basic delay types

6.4 Modelling Delays

6.4.1 Basic Delay Types

As mentioned in chapter 3, there are two basic delay types: *pure delay* and *inertial delay*. Suppose the delay of a digital component is *D*. A pure delay retards a waveform by time *D*, but does not alter it. An inertial delay may alter the shape of a waveform by eliminating the glitches shorter than *D*.

Sometimes, the delay of a component has a more general form. There may exist a threshold $T < D$ such that the component absorbs input pulses whose width is less than *T*. However output follows input if the pulse width is more than *T*. In DILL this is termed *general delay*. In fact, it could be considered as an inertial delay *T* cascaded with a pure delay *D-T*. Figure 39 shows how inputs are related to outputs for different delay types. For clarity, inertial and pure delays, which have be illustrated in chapter 3, are re-drawn in this figure.

The following sections introduce the delay elements that have been included in the DILL library. Although these are components in the sense of building blocks, they do not like most of the components in the library (gates, flip-flops, counters, etc.). Pseudo-components might be a more proper name for them. Unlike the fixed delays *D* discussed above, all delays have a non-deterministic range from *MinDel* (the minimum delay) to *MaxDel* (the maximum delay). This is termed *non-deterministic delay* or *interval time delay* in the sequel. For general delay, *MinWidth* corresponds to the threshold *T*. It is obvious that the assumption of non-deterministic delays is more realistic and flexible than that of fixed delays.

6.4.2 Inertial Delay

The following is a naive attempt at specifying a delay. The example reveals an interesting point related to one aspect of ET-LOTOS semantics: maximal progress on hidden events.

```
process DelayNaive [Ip, Op]
  (MinDel, MaxDel : Time, DataIp, DataOp : Bit) : noexit :=
  Ip ? NewDataIp : Bit;                                          (* new input *)
  DelayNaive[Ip, Op](MinDel, MaxDel, NewDataIp, DataOp) (* continue *)
 []
  [DataIp ne DataOp] ->                                  (* potential output ? *)
  Op ! DataIp {MinDel, MaxDel};               (* output within [MinDel, MaxDel] *)
  DelayNaive [Ip, Op] (MinDel, MaxDel, DataIp, DataIp)            (* continue *)
endproc (* DelayNaive *)
```

The specification uses the ET-LOTOS generalized life reducer to model inertial delay. Outputs happen after the delay*[MinDel, MaxDel]* input has occurred. If another input comes before the delay is due, i.e. the input pulse is less than the delay magnitude, output will not occur. Note that in this specification, the moment when the output *Op* is produced is also determined by the environment, because the delay range is associated with an observable action. But in DILL what should really be specified is that the

delay is decided by the component itself. Moreover if the component with the delay is connected to other components in a larger design, the *Op* port might well be hidden. This would mean that the delay time is exactly *MinDel* instead of being a non-deterministic value, due to the adoption of maximal progress for hidden events in ET-LOTOS. *MinDel* is the earliest moment the hidden *Op* can occur, thus it should occur at that moment.

To overcome the problem, a revised specification is given below:

```
process DelayInertial [Ip, Op]
  (MinDel, MaxDel : Time, DataIp, DataOp : Bit) : noexit :=
  Ip ? NewDataIp : Bit;                                        (* new input *)
  DelayInertial [Ip, Op] (MinDel, MaxDel, NewDataIp, DataOp)
 []
  [DataIp ne DataOp] ->                                  (* potential output *)
  i {MinDel, MaxDel};           (* non-determinisitic delay within [MinDel, MaxDel] *)
  Op ! DataIp {0};                           (* output immediately after delay *)
  DelayInertial [Ip, Op] (MinDel, MaxDel, DataIp, DataIp)
endproc (* DelayInertial *)
```

The specification takes advantage of internal events. The internal event **i** introduces non-deterministic delay, which means the output port can change its value at any time between *MinDel* and *MaxDel*. The exact delay value is determined by the component itself and is not affected by its environment. Moreover, even after the *Op* is hidden in a larger circuit, delay is still non-deterministic since only hidden events are urgent.

As mentioned in section 6.1.1, the internal event *i* has the necessity semantics, in other words *i* is necessarily performed within the time defined by the life reducer. However this property is local and so has no effect on other processes. Especially in a choice context, *i* has no priority over other actions. In the above case, if there is a *Ip* before the i action, *i* can still be prevented. This exactly corresponds to inertial delay, where short pulses are eliminated.

After the occurrence of *i*, *Op* has to happen immediately according to the *0* life reducer. This needs a cooperative environment which is able to participate in *Op* at that moment, otherwise the specification will deadlock. This may indicate either an improper specification or wrong behaviour of the specified circuit.

6.4.3 Pure Delay

Specification of pure delay is done by process forking. A delay component can be regarded as an unbounded first-in-first-out buffer with each output being delayed by a value within [MinDel, MaxDel].

```
process DelayPure [Ip, Op]
  (MinDel, MaxDel : Time, DataIp, DataOp : Bit) : noexit :=
  Ip ? NewDataIp : Bit;                                        (* new input *)
  ([NewDataIp eq DataOp] ->                           (* if no potential output *)
    DelayPure [Ip, Op] (MinDel, MaxDel, NewDataIp, DataOp)
   []
    [NewDataIp ne DataOp] ->                    (* if there is potential output *)
     (
      (i {MinDel, MaxDel};                      (* delay for [MinDel, MaxDel] *)
        Op ! NewDataIp {0};                                        (* output *)
        stop
      )
     |||                                 (* at the same time, process forking *)
      DelayPure [Ip, Op] (MinDel, MaxDel, NewDataIp, NewDataIp)
     )
  )
endproc (* DelayPure *)
```

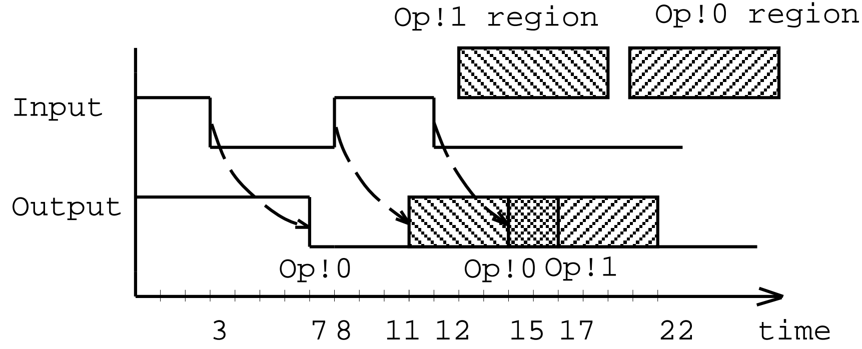


Figure 40: Catch-Up phenomenon with pure delay

In the above specification, every output occurs after delay*[MinDel, MaxDel]* from an input occurring. Before the delay is due, there might be new inputs and consequently new outputs produced. When delay is fixed, although all these outputs are interleaved according to the specification, the proper order of output sequences can still be preserved because the time that each output appears is determined by the delay magnitude. However when delay is assumed to be non-deterministic rather than fixed, the above specification may result in disordered output sequences such as *Op ! 0; Op ! 0; Op ! 1;* . . . , where the second *Op ! 0* overtakes *Op ! 1* and causes the two consecutive *Op ! 0* events. For convenience this is called *catch-up* in the sequel. Figure 40 illustrates that *catch up* occurs if a later input change takes less time to reach the output than an earlier input change. In the figure the delay is between 3 and 9 time units. As one can see, if both events *Op ! 0* and *Op ! 1* happen within the overlapped region then catch up may arise. Suppose the width of a input pulse is *W*. A necessary condition for catch-up to occur is $W \leq MaxDel\text{-}MinDel$.

In fact, catch-up may occasionally emerge in real hardware if delays vary significantly, which is often associated with an unstable environment. However as delays usually fluctuate in a narrow range, the catch-up condition is rarely met in practice. In DILL any delay model that is based on pure delay (e.g. the general delay component to be discussed soon) as well as non-deterministic delay may suffer from this phenomenon. This is not a problem in the inertial delay model since an input change will prevent any pending output; it is therefore not possible to catch up a pending output.

6.4.4 General Delay

As mentioned before, general delay has a threshold *MinWidth*. Input pulses whose width is less than *MinWidth* will be absorbed by the component. They will appear at the output if their width is greater than or equal to *MinWidth*. The general delay element in DILL is specified such that it can model not only a general delay but also inertial and pure delay. This is achieved by choosing appropriate timing parameters. The following specifies the delay component.

```
process Delay [Ip,Op]
  (MinWidth, MinDel, MaxDel : Time, DataIp, DataOp : Bit) : noexit :=
   Ip ? NewDataIp : Bit;                                          (* new input *)
   Delay[Ip,Op] (MinWidth,MinDel,MaxDel,NewDataIp,DataOp)
  []
   [(DataIp ne DataOp)] ->                          (* there is potential output *)
    ([MinWidth lt MinDel] ->                                  (* general delay *)
     (Δ(MinWidth) i;                            (* input holds at least MinWidth *)
      ((i {MinDel − MinWidth, MaxDel − MinWidth};       (* nondeterministic delay *)
        Op ! DataIp {0};                                             (* output *)
        Stop
       |||                                                   (* process forking *)
        DelayAux [Ip,Op] (MinWidth, MinDel, MaxDel, DataIp, DataIp)
       )
```

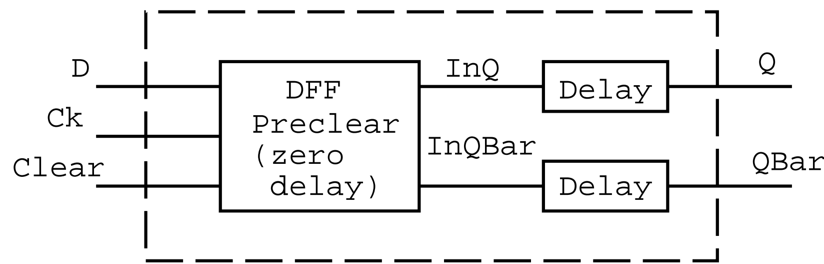


Figure 41: D Flip-Flop with asynchronous Pre-Clear

```
        )
      )
    []
      [MinWidth ge MinDel] ->                                   (* inertial delay *)
        (i {MinDel, MaxDel} ;                            (* nondeterministic delay *)
          Op ! DataIp {0};                                             (* output *)
          DelayAux [Ip,Op] (MinWidth, MinDel, MaxDel, DataIp, DataIp)
        )
    )
endproc (* Delay *)
```

This specification is essentially the combination of those for inertial and pure delays. When *MinWidth* $\geq$ *MinDel* it is identical to the inertial delay specification. When *MinWidth* $<$ *MinDel*, it is the case of pure delay. The component first waits for *MinWidth*, during which input *Ip* has a chance to prevent output, eliminating glitches shorter than *MinWidth*. Then it enters the pure delay phase, which is also specified by process forking.

Different combinations of the time parameters bring different kinds of delay components:

$0 < MinWidth < MinDel \leq MaxDel < Inf$ describes general delay.

$MinWidth = 0, MinDel \leq MaxDel < Inf$ is the case of pure delay. The difference between general delay and pure delay is that in the latter *MinWidth* is zero so that the component does not absorb a narrow pulse.

$0 \leq MinDel \leq MaxDel < Inf, MinWidth > MinDel$ is the case of inertial delay. It applies if the threshold *MinWidth* is greater than *MinDel*. *MinWidth* is often set to *Inf* for inertial delay.

$MinDel = 0, MaxDel = Inf, MinWidth > 0$ is equivalent to the untimed delay component specified in section 3.2. Usually *MinWidth* is given the value *Inf*.

6.4.5 Delay Components for Higher Level Specifications

In higher level specifications of components, delays from several inputs to the same output may well different. The delay components specified above assume the same range of delay for all inputs to the same output, which turns out to be unrealistic when used with higher level components. For example, consider a D (delay) flip-flop with asynchronous pre-clear.[6] Suppose the delay from clock (*Ck*) to outputs (*Q* and *QBar*) is 20–30 ns, while the delay from the asynchronous clear to the output being reset could be as little as 10–15 ns. Forcing a common range for them is hence unreasonable.

A delay component for higher level specifications is thus required. When a change turns up at an input of the delay component (in this example, *InQ*), if there is no indication of the source of the change (a clock transition or a clear), the delay component will have no idea about which delay value should be

[6]This is a one-bit memory element that stores data *D* under the control of a clock signal *Ck*. Its outputs *Q* and *QBar* (negated output) can be reset with a clear signal at any time irrespective of the clock. If a clear is not being requested (value 1), after the positive transition of *Ck* the input data will appear at the output after some delay. If a clear is requested (value 0), the output will be cleared asynchronously no matter what the level of the clock is.

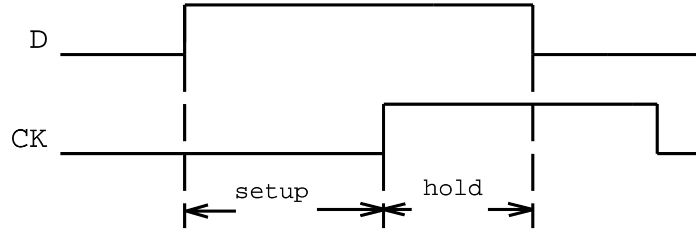


Figure 42: Setup and hold times for D Flip-Flop

applied (20–30 ns or 10–15 ns). To make things clear, it requires the functional specification to declare delay magnitudes when outputs are offered. For example, after *Clear* is reset to *0*, the flip-flop must offer *InQ ! 0 ! 10 ! 15* to ensure that on this occasion *InQ* should be delayed for 10 to 15 ns. If *InQ* is changed because of a clock transition, the flip-flop must offer *InQ ! 0 ! 20 ! 30*.

As one might expect, because different delay values are applied to one delay component, the catch-up phenomenon will arise in a pure delay specification, even if each delay value is fixed. Delay components for higher level specifications in the DILL library are therefore based only on inertial delay.

6.5 Modelling Timing Constraints

Timing constraints in DILL are used to check if inputs of a component satisfy some conditions. There are various common timing constraint such as *setup, hold, pulse width* and *period* etc.

Setup and hold times are always associated with flip-flops. For a D (delay) flip-flop, setup time is the time interval between a change on input *D* and the trigger that stores this data (e.g. a positive-going transition of the clock *Ck*). The data signal must then remain stable for a minimal time interval if correct operation of the flip-flop is to be guaranteed. For a flip-flop, the hold time is the interval in which input data must remain unchanged after triggering by the clock. Again, this minimum must be respected for correct operation. A timing diagram showing setup time and hold time is given in figure 42.

The setup time constraint is specified as follows, supposing that the active clock transition is positive-going. As explained in section 6.2, an additional gate *Err* is introduced to detect violation of the constraint and to simplify specification under erroneous inputs. After input on *D* takes place, it is necessary to notice the next event and the time it appears. If within the *setup* time there are no events at all, this *D* passes the check. If a negative clock transition shows up, the time is recorded so that further events can be checked in the remainder of the setup time. If a positive transition of clock signal comes within the checked time, the *Err* gate has to be used to show that a violation was detected. It is also possible that several *D*s come in a string. In this case, the moment that the last *D* happens is used as the start point of the setup time:

```
process SetupDel [D, Ck, Err] (SetupTime : Time) : noexit :=
   D ? NewDataIp: Bit;                                        (* new data input *)
   AfterD [D, Ck, Err] (SetupTime, SetupTime)               (* check setup time *)
[]
   Ck ? NewCk : Bit;                                         (* new clock input *)
   SetupDel [D, Ck, Err] (SetupTime)                   (* no setup time to check *)
endproc (* SetupDel *)

process AfterD [D, Ck, Err] (SetupTime, SetupRem : Time) : noexit :=
   Δ(SetupRem) i;                                  (* no events during SetupTime *)
   SetupDel [D, Ck, Err] (SetupTime)                     (* go to the next round *)
[]
   Ck ? NewCk : Bit @ t;                                     (* new clock input *)
   (
      [NewCk eq 0] ->                                  (* negative-going clock? *)
         AfterD [D, Ck] (SetupTime, SetupTime − t) (* check remaining setup time *)
   []
```

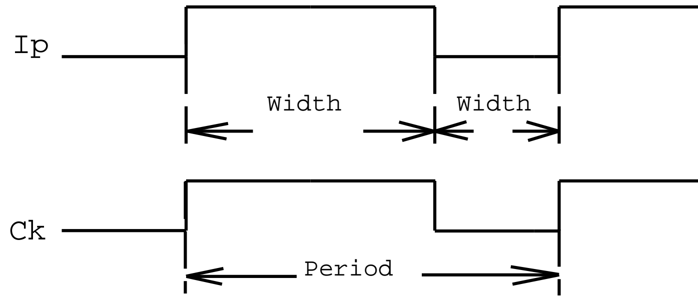


Figure 43: Width and Period timing constraints

```
        [NewCk eq 1] ->                                   (* positive-going clock *)
          Err ! SetupError;                               (* setup time violated *)
          SetupDel [D, Ck, Err] (SetupTime)               (* go to the next round *)
      )
    []
      D ? NewDataIp: Bit;                                 (* new data input *)
      AfterD [D, Ck, Err] (SetupTime, SetupTime)          (* set the new start point *)
    endproc (* AfterD *)
```

The hold time constraint is specified in a very similar way. There are also other timing constraints in the DILL library. For example, the width timing constraint defines the minimum *width* that an input pulse should have. The period timing constraint is the minimum *period* for an input signal, especially clock signals. Figure 43 illustrates the two constraints.

6.6 Case Study: A 2-to-1 Multiplexer

In this section, a small circuit (a 2-to-1 multiplexer) is specified and validated. The validation goal is to examine if there are hazards in the design, which is usually done by analysing delays of components. Of course a DILL specification can also be used for other purposes such as deciding clock periods for synchronous circuits, calculating timing performance, etc. But because of the limitation of tools, only small examples are studied so far.

A 2-to-1 multiplexer has two data inputs *A* and *B*, a selection input *S* and an output *C*. The behaviour is such that if the selection input is 0, the data at *A* will appear at *C* after some delay. Alternatively if the selection input is 1, the data at B will appear at *C*. The delays used in the example are inertial, mainly because they are easy to handle and are more general than pure delay.

The multiplexer is specified at two levels. The higher level specifies the required behaviour and timing performance. The lower level specifies the structure of the component by connecting basic logic gates. The lower level implements the higher level. The timed specifications are analysed through simulation and testing.

6.6.1 TE-LOLA

As standard LOTOS is untimed, there has been little tool support for timed extensions of LOTOS. The only tool available is TE-LOLA, which supports TE-LOTOS (Time Extended LOTOS [RQ96]). Although ET-LOTOS and TE-LOTOS adopt different semantic models, the equivalence between them has been established [LR95]. It is therefore possible to translate ET-LOTOS specifications into TE-LOTOS ones. Because of their similarity, the translation is always possible although some subtle differences need attention. For example, **i** $\{d\}$ in ET-LOTOS means **i** will happen non-deterministically between *0* and *d* time units, but in TE-LOTOS it means that **i** will occur at exactly time *d*. The correct translation should be **i** $\{0..d\}$ in TE-LOTOS. In order to avoid confusion, the following specifications will still use ET-LOTOS syntax, although the actual analysis was made with TE-LOTOS.

The validation functions of TE-LOLA are simulation and testing, which are both exploited in this case study.

6.6.2 Behavioural Specification and Validation

Behavioural specification of the 2-to-1 multiplexer uses two processes, one defines the functionality and the other defines the delay type and magnitude. The higher level specification of the multiplexer is specified simply by composing these two parts. *Multiplexer[A, B, S, InC] (0, 0, 0, 0)* is the process for zero delay multiplexer with every ports initially being at *0*. DelayInertial [InC, C](10, 15, 0, 0) is the instantiation of the inertial delay specified in section 6.4.2, with the delay in the range 10 to 15 time units.

```
process Multiplexer [IA, IB, IS, OC] (dtIA, dtIB, dtIS, dtOC) : noexit :=
  IA ? newdtIA : Bit [newdtIA ne dtIA] ;                    (* IA is changed *)
  Multiplexer [IA, IB, IS, OC] (newdtIA, dtIB, dtIS, dtOC)
 []
  IB ? newdtIB : Bit [newdtIB ne dtIB] ;                    (* IB is changed *)
  Multiplexer [IA, IB, IS, OC] (dtIA, newdtIB, dtIS, dtOC)
 []
  IS ? newdtIS : Bit [newdtIS ne dtIS] ;                    (* IS is changed *)
  Multiplexer [IA, IB, IS, OC] (dtIA, dtIB, newdtIS, dtOC)
 []
  let newdtOC : Bit = (dtIA and not(dtIS)) or (dtIB and dtIS) in
   ( [newdtOC ne dtOC ] -> OC ! newdtOC {0};      (* output change immediately *)
      Multiplexer [IA, IB, IS, OC] (dtIA, dtIB, dtIS, newdtOC)
   )
endproc (* Multiplexer *)

hide InC in                                           (* internal gate to delay *)
  Multiplexer [A, B, S, InC] (0, 0, 0, 0)            (* multiplexer instance *)
|[InC]|                                                  (* sync with delay *)
  DelayInertial[InC, C] (10, 15, 0, 0) )                  (* delay instance *)
```

The behavioural specification was validated by the simulation and testing functions of TE-LOLA. The aim is to ensure that the specification is as expected. As is well known, both simulation and testing are not exhaustive validation. This is especially true for timed specifications with a dense domain: an event can take place at any time so there is no way to give all possible execution paths. The strategy for validation is to focus on representative `states', for example *A ! 1 {0}; B ! 1 {0}; S ! 1 {0}* is used to stand for the situations where all inputs changes to *1*. They may change to *1* at different times in different orders, but it is impossible to list all of the situations. There are three inputs here so there are 8 input `states' in total. Simulation is done by randomly choosing these input states one by one to see if the outputs are right. The recorded simulation paths can also be used as the criterion when the lower level specification is validated.

Testing is a more efficient and reliable method compared to simulation because one test case can cover many simulation paths. In TE-LOLA testing is done by composing test processes in parallel with the original specification. Each test process is a test case. If the test process can be followed for all executions of the composed specification, the result of testing is *must pass*. If the test process can be followed only for some executions, the result is *may pass*. Otherwise the test is considered to be *rejected*.

For this example, testing can be conducted in two steps. First, it should be made sure that from the initial state there is no problem to move to any other states. Seven test processes are defined corresponding to moving to the seven states other than the initial one. For instance, after moving to the state *A=1, B=1, S=0* the output should be *C=1* after 10 or 15 time units. Second, it is necessary to check that after the first correct movement, the specification can always change to any other state correctly. 56 (8×7) processes are designed because each one of the 8 states can move to the other 7 states. If it is assumed that the multiplexer has iterative behaviour (this is actually a testing hypothesis), the above 63 (7 + 56) test processes should have satisfactory coverage. In fact, TE-LOLA supports executing all test cases from a batch file, thus a single run can obtain all the test results.

The higher level specification is proven to be correct according to simulation and testing.

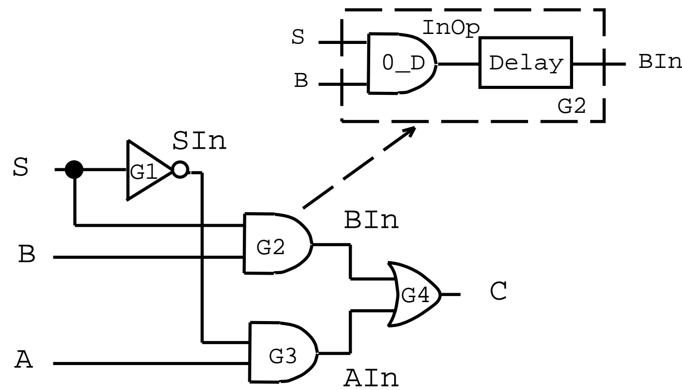


Figure 44: Structure of 2-to-1 multiplexer as timed logic gates

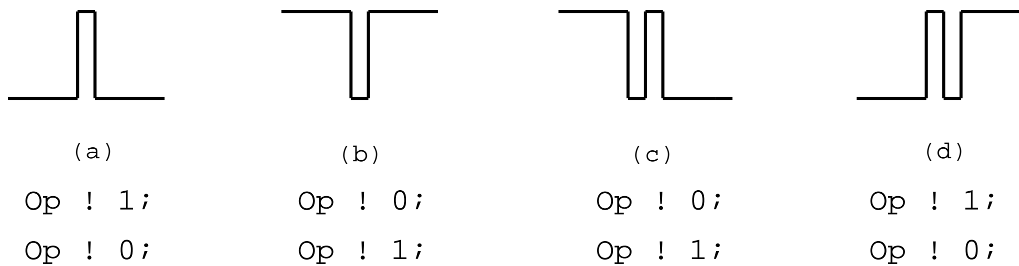


Figure 45: Hazards and their LOTOS specifications

6.6.3 Structural Specification and Validation

The structure of the 2-to-1 multiplexer is shown in figure 44.[7] The logic gates in the diagram are timed gates. Each of them consists of zero-delay logic and a delay component. The inset in the figure shows the structure of the *and* gate *G2*; *0_D* in the figure means zero delay. Other gates have the same kind of structure. All gates are supposed to have a fixed delay which is 5 time units in this example. The design of the multiplexer is `classical' and can be found in textbooks like [Kli83]. However, as will be seen later this design contains timing hazards.

Hazards are unwanted transitions that appear on the outputs of digital circuits in response to the changes on inputs. For example, suppose that the output should stay the same (e.g. *1*) after an input changes from state I_1 to I_2. But what happens in an actual implementation is that the output changes from *1* to *0* and then back to *1* after the input. The consecutive unwanted transitions *1* to *0* and *0* to *1* are regarded hazards. Figure 45 illustrates kinds of common hazards in circuits and their corresponding LOTOS specifications. Cases (a) and (b) are called *static-0* and *static-1* hazards respectively, while (c) and (d) are called *dynamic* hazards.

The simulation paths and test processes for validating the higher level specification are re-used to analyse the lower level design. Below is one of the test cases which aims to detect if there is a hazard when the circuit moves from state 111 to 110 (*A=1, B=1, S=1* to *A=1, B=1, S=0*). This test case should have been rejected if there were no hazard, however the result is *may pass* indicating a static hazard exists during this transition.

```
process Test111_110Hazard [A, B, S, C, OK] : noexit :=
    A ! 1 {0}; B ! 1 {0}; S ! 1 {0};                    (* change to state 111 *)
    C ! 1 {10, 15}                                                  (* output 1 *)
    S ! 0 {2};                                          (* change to state 110 *)
     ( C ! 0 {10, 15};        C ! 1;                                  (* hazard *)
```

[7] The types of the gates are omitted. The triangle with a circle is an inverter, the `D' shapes are *and* gates, the shield shape is an *or* gate.

Transition	Type of Hazard	Number of Changed Inputs
000 to 101	static-0	2
010 to 101	static-0	3
011 to 100	static-1	3
011 to 110	static-1	2
111 to 100	static-1	2
111 to 110	static-1	1

Table 6: Hazards in the 2-to-1 Multiplexer

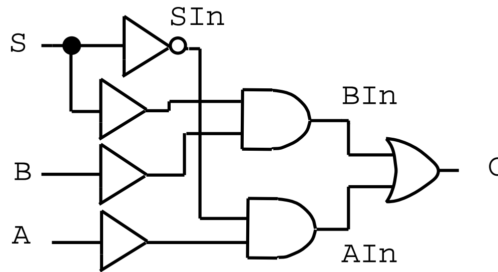


Figure 46: The hazard-free Multiplexer

OK; **stop**
)
endproc (* Test111_110Hazard *)

Evaluating all test processes shows that 6 of them pass the test (when they should have been rejected). Table 6 lists these transitions and the corresponding hazards. The test results indicate that when the delays of each gate are fixed, the circuit exhibits static hazards. One of the hazards happens when there is a single input change; the others occur when more than one input changes simultaneously.

By analysing a passed test sequence it is possible to discover the cause of the hazard: the inputs follow different lengths of path to reach the output. Figure 46 is a very simple solution to the problem (though it may not be realistic): three redundant delay components are used to guarantee that each input-output path is exactly three gate delays. It is obvious that each delay component should have the same delay value as the basic logic gates used in the design. In practice, they could be repeaters or some other digital components which have the desired delay value. This revised design is proven to be correct by the same testing procedure.

Finally, it should be pointed out that the original design of the multiplexer is usually used in synchronous circuits, which means the hazards discovered will have no influence when clock cycles are slow enough (for example, longer than 15 time units). Apparently, the design must not be used in building asynchronous circuits due to these hazards discovered.

6.7 Related Work

Industrial HDLs (hardware description languages) such as VHDL ([IEE93]) and Verilog ([IEE95]) support simple timed specifications of digital circuits. Among various timing characteristics, only fixed inertial and pure delays are specifiable in these two languages. To improve timing accuracy, OVI (Open Verilog International) adopted SDF (Standard Delay Format) for representation and interpretation of timing data at any stage of circuit design. A wide variety of timing data can be specified in great detail in SDF. For example, delays are allowed to be non-deterministic and different kinds of delays, such as port delays, path delays and interconnect delays are modelled. Dozens of timing checks (setup timing, hold timing, ...) and timing environments (path constraints, period constraints, ...) are also supported. At the specification

stage, SDF files can be used as supplements to VHDL or Verilog program to introduce more precise timing information. SDF will become a future IEEE standard [IEE99].

Formally specifying and analysing timed circuits has attracted reasonable attention only after the resurgence of asynchronous circuit design. For synchronous circuits, quantitative timing is normally abstracted in both specification and validation. Although it is also possible to avoid timing in delay-insensitive (DI) or speed-independent (SI) circuits, timed asynchronous circuits can be significantly smaller and faster than those assuming unbounded delays.

In [HB97], CSP was used to specify asynchronous circuits. After a CSP program was transformed into a safe Peri Net, non-deterministic time delays (called interval delays in that paper) were annotated on the places of the nets. An algorithm was then developed to determine the extreme separation in time between two communication actions of the CSP program. The specification aspect of this methodology is apparently poor; the authors' main attention was to develop an efficient analysis algorithm.

In [MY96], timed automata ([AD94]) were employed to specify the behaviour of MOS transistors directly, as well the behaviour of inputs and environment. A circuit is then the composition of these automata. TCTL (Timed Calculation Tree Logic [ACD90]) is the formalism for specifying properties. Verification is done by model checking of the TCTL formula against the timed automata, which is performed automatically by the tool KRONOS ([DOTY96]).

[MP95] utilized the same tool to analyse gate level asynchronous circuits. The authors developed a formalism called *timed boolean functions* to specify circuits. Each gate is specified by a function, and a circuit is simply a set of all gates. Each function is actually the combination of two sub-functions: one for the functionality of the gate, the other for the delay associated with the gate. Inertial delay (termed latency in the paper) and pure delay (termed ideal in the paper) are modelled, although only inertial delays are really used in circuit specifications. The authors proved that each circuit modelled by *timed boolean functions* could be transformed into an equivalent timed automaton, allowing analysis based on verification tools such as KRONOS.

The process algebra CIRCAL has also been used in real time systems ([CKM98a, CKM98b]). Unlike LOTOS, CIRCAL itself is not extended for this purpose. Instead, an action *t* is regarded as a global tick, and specific processes are defined with respect to *t* to model various delays. The approach becomes very complicated when a dense time domain is considered. Except for a global clock, each process should have its own local clock, and there are also local clocks for every two processes which have interactions. All local clocks can be set and reset when necessary, which is the responsibility of specifiers. Moreover, in order to keep each local clock progressing at the same rate, there should be a process to control the progress for every two local clocks. The advantage of the CIRCAL methodology is that the language itself needs no extension, neither do the tools supporting it. The disadvantages are that the burden of maintain timing mechanisms is actually moved to circuit specifiers.

By using ET-LOTOS, the thesis can specify timed circuits at various level of abstraction, i.e. not just gate level or whatever. Specifications are intuitive and concise, thanks to the timed semantics of ET-LOTOS. Because specifications include the most important timing characteristics, namely delays and timing constraints, various properties can be analysed. The main impediment at the moment is that because ET-LOTOS has not yet become an ISO standard, few tools support it. This will be overcome in the near future with the appearance of the new standard E-LOTOS.

6.8 Conclusion

This chapter has used ET-LOTOS to specify timed circuits. Two important timing characteristics in digital circuits, namely delays and timing constraints, have been identified. A timed component is modelled as a zero delay part followed by a delay component. If necessary, timing constraints are used to guard the inputs to ensure that input timing conditions are respected. The model is compositional, and has the nice property that untimed components are just special cases of timed ones. Various delays and timing constraints are provided by the DILL library. It should be pointed out that when pure delays are associated with non-deterministic values, DILL specification will suffer from the catch up phenomenon, which might not be a realistic representation of real hardware.

Timed specification can serve as the basis of various analyse. For example, it can be used to check whether timing requirements on a digital design are respected. This can be done by using the timing

constraint components. As in the multiplexer example, it is also valuable in discovering potential timing errors like hazards. It can also be used to analyse the timing properties of a logic design such as its minimal and maximal delays.

To gain these benefits really needs the help of tools. In the case study of the multiplexer, test cases were generated manually, It would be ideal if all tests could be generated automatically. Testing theory of ET-LOTOS has been established in [L9́7]. What is missing is conformance relations and test generation algorithms, which require further theoretical investigation. Future work based on this thesis is to formally verify timed circuits. One possibility is to used the tool KRONOS, which checks if the system described by a timed automaton satisfies a requirement expressed as a formula of TCTL. A method for transforming ET-LOTOS specifications to timed automata has already been implemented in [DOY95, Her97]. Verification of a timed DILL specification may thus be possible.

7 Conclusion

This chapter concludes this thesis. Because the former chapters have already contained individual summaries, this chapter focuses on the main contributions of the thesis, then provides suggestions for future work.

7.1 Main Contributions

The thesis uses the formal language LOTOS to specify and analyse digital circuits. It investigates the topic from several different but closely related aspects, namely specification, verification, testing, and timing.

The underlying modelling approach of DILL was developed in [TS94], which mainly included the models for signals, wires, components, the connection of components, and a specification of basic logic gates which has been presented again in section 3.2 . All the larger circuits in [TS94] were built from basic logic gates. In order to support the design procedure used in industry, the thesis identified that it is necessary for DILL to specify circuits at different levels of abstraction. Therefore not only structural specifications should be supported, but also behavioural specifications. Many common components were subsequently provided with behavioural specifications. At the same time, the data type *BitArray* was defined to aid higher level behavioural specification.

The basic modelling approach was then applied in specifying synchronous and asynchronous circuits. Since circuit structure can be specified routinely, chapter 3 focused on the behavioural specifications. Apart from providing specifications of common building blocks of both synchronous and asynchronous circuits, this chapter resulted in several important observations: 1) The same component may have different models in different kinds of circuits. 2) The same component may have different model when different verification purposes are required. 3) When LOTOS events models signal transitions, as in the case of modelling asynchronous circuits, the behaviour of a LOTOS specification may not represent its corresponding circuit very well since LOTOS does not differentiate inputs and outputs. 4) To overcome the problem in 3), *input receptive* and *input quasi-receptive* specifications should be employed.

Chapter 4 provided the approach of verifying digital circuits specified in LOTOS. Unlike most hardware verification approaches and tools, DILL supports the three conventional formal verification tasks, namely *requirements capture, implementation verification* and *design verification*. Basically implementation verification is achieved by comparing the relations between LTSs, and the other two tasks can be done by model checking temporal logic formulae. The chapter reveals that the existing relations characterising the relationships between two LTSs are not suitable for implementation verification of asynchronous circuits. Two relations are therefore defined to solve the problem. The relations consider the difference between inputs and outputs in hardware, and provides intuitive criteria for the correctness of asynchronous circuits. A verifier *VeriConf* was implemented for checking the relations. The case study in the chapter discovered a bug in the design of a Bus Arbiter, a benchmark circuit which has been verified by many researchers.

Chapter 5 explores a new direction of applying formal methods in digital circuit design. Here, test vectors (called test cases in the community of formal verification) are generated automatically from the behavioural specification of a circuit, then fed into a commercial VHDL simulator to simulate the design of the circuit. The method alleviates the state explosion problem by avoiding generating the state space of circuit designs, which are usually much larger than their corresponding behavioural specifications. The approach is helpful in finding subtle bugs which cannot be detected by formal verification, since a simulation model of a circuit (e.g. The VHDL description of the circuit) is usually closer to real hardware than a formal model. A test generator has been implemented, which guarantees to cover all possible transitions of the state graph of the behavioral specification. The case studied in the chapter discovered a bug in another benchmark circuit, namely the BlackJack Dealer. This circuit is also studied by other researchers using formal verification. But this bug, which is related to the wrong timing in the design, might never be discovered by formally verifying a model which does not contain timing information.

Chapter 6 used ET-LOTOS to write circuit specifications which contain quantitative timing magnitudes. This chapter identified the important timing characteristics in digital circuits, namely *timing constraints* and *delays*, then specified them in ET-LOTOS. The model for timed components and circuits was also established. Unlike most formal hardware specification and verification approaches which ignore quantitative timing, this chapter is a new attempt to address the issue. Timed specifications of digital circuits provide

the basis for thoroughly analysing circuit behaviour which is sensitive to timing, and can also be used in evaluating the speed performance of digital circuits.

One of the motivations of the thesis was to examine the possibility of applying LOTOS outside its traditional area. Through the investigation, it was discovered that LOTOS is suitable for specifying digital circuits, including both synchronous and asynchronous circuits. Compared with other formalisms used in hardware specifications, LOTOS has a clear advantage in higher level specifications, such as at the system level or the algorithm level. Its status as international standard helps to reduce the efforts in developing analysis tools greatly. The thesis demonstrated that most general LOTOS tools can be used in analysing DILL specifications directly. Even when such tools are not available, customized tools can be quickly developed by using the programming interface provided. Thanks to these well-developed tools, the thesis was able to span several different aspects of validating digital circuits in a relatively short time.

The thesis also identifies some limitations when LOTOS is used in the area of digital circuits. The overhead of the language makes it more difficult for formal verification. The state space generated directly from a LOTOS specifications is normally much larger than the real state space of a circuit. Current LOTOS tools can only generate the larger LTSs then minimise them to smaller ones. However if the initial state spaces are not able to be generated in the first place, minimisation cannot be applied. Although this is really the problem of verification tools rather than that of the language, it restricts DILL from analysing larger circuits. The breakthrough might lie in using the syntax based verification approach; hopefully there are already such theories developed for LOTOS such as in [Kir94, ST97, MT94].

There is also a gap between the process communication scheme adopted in LOTOS and the communication scheme between real hardware components. This happens when LOTOS events model signal transitions. In LOTOS the communication of processes is achieved by the synchronisation of common events. Processes can refuse events when they are not ready to accept them. But in real hardware, input signal transitions can never be refused by components. As a result of the difference, behaviour which happens in real hardware might not be represented by their LOTOS model. To accurately model circuit behaviour, the thesis suggest *input receptive* and *input quasi-receptive* specifications. It should be pointed out that these specifications are usually more difficult to write, and that input quasi-receptive specifications usually result in a larger state space which makes verification more difficult.

The second motivation of the thesis was to provide theories and tools to aid designing correct hardware. DILL advocates the component-based specification style which emphasizes the re-use of trusted components. It comes with a comprehensive library which contains the validated specifications of commonly used digital components. Using these library components will help to reduce errors in specifications. DILL supports implementation verification, as shown in the case study of the bus arbiter, it is a complementary approach to design verification and should be used when possible to detect as many bugs as possible in designs. DILL also explores the new area of combining formal methods with the traditional simulation approach. The results of the exploration show that LOTOS testing theory can be successfully employed in the area. Two tools were developed along the theoretical investigations, namely *VeriConf* and *TestGen*, which help to support the case studies in the thesis, and which as well can be employed to validate other hardware circuits.

7.2 Future Work

With E-LOTOS becoming the new standard of LOTOS, new investigation should be made into using E-LOTOS to specify and analyse digital circuits. The investigation will provide feedback of advantages and limitations on the language, which should be of interest to the language developers since E-LOTOS is still in the course of standardization. It will also benefit the community of hardware designers for the following reasons:

- E-LOTOS adopts many feature of common imperative languages so that it is more user friendly than LOTOS. DILL specifications will be easier to write if E-LOTOS is to be used as the underlying formalism.
- The current DILL approach provides limited support for analysing timed specification, due to the lack of proper tools. After E-LOTOS becomes an international standard, more tools will be developed to

support it. It is hoped that the new tools will make it possible to analyse the timing characteristics of practical circuits.

Another immediate research direction is the test selection problem. In chapter 5, a test suite is guaranteed to cover all the transitions of the state space of a specification, but how much it covers the whole behaviour of the specification is unknown. Moreover, when the circuits are relatively complex, the size of the test suite might be very large resulting in considerable simulation run-time. The methodologies of test selection advocated in [BTV91, ACV93, CG97] might help. Heuristics related to circuits should also be explored, which is extremely useful for testing important parts of a circuit, or the parts that are subject to errors.

References

[ACD90] R. Alur, C. Courcoubetis, and David L. Dill. Model-checking for real-time systems. In *Proc. 5th Symposium on Logic Computer Science*, pages 414–425, 1990.

[ACV93] J. Alilovic-Curgus and Son T. Vuong. A metric based theory of test selection and coverage. In André A. S. Danthine, Guy Leduc, and Pierre Wolper, editors, *Proc. Protocol Specification, Testing and Verification XIII*, pages 289–304. North-Holland, 1993.

[AD94] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, April 1994.

[BBM94] Peter A. Beerel, Jerry R. Burch, and Teresa H.-Y. Meng. Sufficient conditions for correct gate-level speed-independent circuits. In *Advanced Research in VLSI*, pages 33–43, November 1994.

[BCM+92] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, June 1992.

[Ber91] G. Bernot. Testing against formal specification: A theoretical view. In Samson Abramsky and Thomas S. E. Maibaum, editors, *TAPSOFT'91* , volume 494 of *Lecture Notes in Computer Science*, pages 99–119. Springer-Verlag, Berlin, Germany, 1991.

[BFG+91] A. Bouajjani, J. Cl. Fernandez, S. Graf, C. Rodriguez, and J. Sifakis. Safety for branching time semantics. In J. Leach Albert, B. Monien, and M. Rodríguez, editors, *Automata, Languages and Programming*, volume 510 of *Lecture Notes in Computer Science*, Berlin, Germany, 1991. Springer-Verlag.

[BGMW95] Howard Barringer, Graham Gough, Brian Monahan, and Alan Williams. Formal support for the ELLA hardware description language. In Paolo E. Camurati and Hans Eveking, editors, *Correct Hardware Design and Verification Methods, CHRME'95* . Springer-Verlag, 1995.

[BH+96] Robert K. Brayton, Gary D. Hachtel, et al. VIS : A system for verification and synthesis. In T. A. Henzinger and R. Alur, editors, *Computer Aided Verification 1996*, volume 1102 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.

[BM91] Peter Beerel and Teresa Meng. Semi-modularity and self-diagnostic asynchronous control circuits. In Carlo H. Séquin, editor, *Advanced Research in VLSI*, pages 103–117. MIT Press, March 1991.

[Bri87] Ed Brinksma. On the existence of canonical testers. Technical Report INF-87-5, University of Twente, Enschede, Netherlands, January 1987.

[Bri88] Ed Brinksma. A theory for the derivation of tests. In Sudhir Aggarwal and Krishan K. Sabnani, editors, *Proc. Protocol Specification, Testing and Verification VIII*. North-Holland, Amsterdam, Netherlands, June 1988.

[Bry92] R. E. Bryant. Symbolic Boolean Manipulation with Ordered Binary Decision Diagrams. *CM Computing Surveys*, 24(3):293–318, September 1992.

[BTV91] Ed Brinksma, Jan Tretmans, and Louis Verhaard. A framework for test selection. In Bengt Jonsson, Joachim Parrow, and Bjørn Pehrson, editors, *Proc. Protocol Specification, Testing and Verification XI*, pages 233–248, Amsterdam, Netherlands, 1991. North-Holland.

[BZ97] J. A. Brzozowski and H. Zhang. Delay-insensitivity and semi-modularity. Technical Report CS-97-11, Dept. of Comp. Science, Univ. of Waterloo, March 1997.

[CES86] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite state concurrent systems using temporal logic specification. *ACM TOPLAS*, 8, 1986.

[CG97] Olivier Charles and Roland Groz. Basing test coverage on a formalization of test hypotheses. In M. Kim, S. Kang, and K. Hong, editors, *International Workshop on Testing Communicating Systems X*, pages 109–124. Chapman-Hall, London, UK, 1997.

[CKM98a] A. Cerone, D.A. Kearney, and G.J. Milne. Integrating the verification of timing, performance and correctness properties of concurrent systems. In *International Conference on Application of Concurrency to System Design (CSD'98)*, pages 109–119. IEEE Computer Society Press, 1998.

[CKM98b] A. Cerone, D.A. Kearney, and G.J. Milne. Verification of timing, performance and correctness properties of a four-phase asynchrounous micropipeline. Technical Report CIS-98-002, University of South Australia, 1998.

[CT97] Graham Clark and George Taylor. The verification of asynchronous circuits using CCS. Technical Report ECS-LFCS-97-369, Computer Science Department, University of Edinburgh, 1997.

[CW96] E. Clarke and J. Wing. Formal methods: State of the art and future directions. Technical Report CMU-CS-96-178, CMU, August 1996.

[DCS93] A. Davis, B. Coates, and K. Stevens. The Post Office experience: Designing a large asynchronous chip. In *Proc. Hawaii International Conf. System Sciences*, volume I, pages 409–418. IEEE Computer Society Press, January 1993.

[Dea92] M. E. Dean. *STRip: A self-timed RISC processor architecture*. PhD thesis, Stanford University, 1992.

[DH84] R. De Nicola and M. C. B. Hennessy. Testing equivalences for processes. *Theory of Computer Science*, pages 83–133, 1984.

[Dil89] David L. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits*. ACM Distinguished Dissertations. MIT Press, 1989.

[Dil98] David L. Dill. What's between simulation and formal verification. In *ACM/IEEE Design Automation Conference'98*, 1998.

[DN95] Al Davis and Steven M. Nowick. Asynchronous circuit design: Motivation, background, and methods. In Graham Birtwistle and Al Davis, editors, *Asynchronous Digital Circuit Design*, Workshops in Computing, pages 1–49. Springer-Verlag, 1995.

[DOTY96] C. Daws, A. Olivero, S. Tripakis, and S. Yovine. The tool KRONOS. In B. Mounier, J. L. Albert, and M. Rodriguez, editors, *Proc. Hybrid System III*, volume 1066 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, Germany, 1996.

[DOY95] C. Daws, A. Olivero, and S. Yovine. Verifying ET-LOTOS programs with KRONOS. In Richard L. Tenney, Paul D. Amer, and M. Ümit Uyar, editors, *Proc. Formal Description Techniques VI*, pages 227–242. Chapman-Hall, London, UK, 1995.

[DV90] Rocco De Nicola and Frits Vaandrager. Action versus state based logics for transition systems. In *Semantics for Systems of Concurrent Processes*, volume 469 of *Lecture Notes in Computer Science*, pages 407–419. Springer-Verlag, Berlin, Germany, 1990.

[Ebe91] Jo Ebergen. A formal approach to designing delay delay-insensitive circuits. *Distributed Computing*, pages 107–119, 1991.

[EJ72] J. Edmonds and E. L. Johnson. Matching, Euler tours and the Chinese postman. *Mathematical Programming*, 5:88–124, 1972.

[EM85] Hartmut Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1*, volume 6 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, Berlin, Germany, 1985.

[ESB95] Jo C. Ebergen, John Segers, and Igor Benko. Parallel program and asynchronous circuit design. In Graham Birtwistle and Al Davis, editors, *Asynchronous Digital Circuit Design*, Workshops in Computing, pages 51–103. Springer-Verlag, 1995.

[FGK+96] Jean-Claude Fernández, Hubert Garavel, Alain Kerbrat, Radu Mateescu, Laurent Mounier, and Mihaela Sighireanu. CADP (CÆSAR/ALDÉBARAN Development Package): A protocol validation and verification toolbox. In Rajeev Alur and Thomas A. Henzinger, editors, *Proc. 8th. Conference on Computer-Aided Verification*, number 1102 in Lecture Notes in Computer Science, pages 437–440. Springer-Verlag, Berlin, Germany, August 1996.

[FGM+92] Jean-Claude Fernández, Hubert Garavel, Laurent Mounier, A. Rasse, and Carlos Rodriguez. A toolbox for the verification of LOTOS programs. In *Proc. 14th. International Conference on Software Engineering and its Applications*, pages 246–259, May 1992.

[FJJV96] J. C. Fernandez, C. Jard, T. Jéron, and C. Viho. Using on-the-fly verification techniques for the generation of test suites. In Rajeev Alur and Thomas A. Henzinger, editors, *Computer Aided Verification'96*, volume 1102 of *Lecture Notes in Computer Science*, pages 348–359. Springer-Verlag, Berlin, Germany, 1996.

[FK97] K. Fisler and Robert P. Kurshan. Verifying VHDL designs with COSPAN. In *Formal Hardware Verification Methods and Systems in Comparison*, volume 1287 of *Lecture Notes in Computer Science*, pages 206–247. Springer-Verlag, Berlin, Germany, 1997.

[FPJ+94] S.B. Furber, P.Day, J.D.Garside, N.C.Paver, S.Temple, and J.V.Woods. The design and evaluation of an asynchronous microprocessor. In *In Proceeding of the IEEE International conference on Computer Design*, pages 217–220. IEEE computer Society Press, October 1994.

[GBMN94] Ganesh Gopalakrishnan, Erik Brunvand, Nick Michell, and Steven Nowick. A correctness criterion for asynchronous circuit validation and optimization. *IEEE Transactions on Computer-Aided Design*, 13(11):1309–1318, November 1994.

[GDKW92] A. A. Ghosh, S. Devadas, K. Keutzer, and J. White. Estimation of average switching activity in combinational and sequential circuits. In *the 29th Design Automation Conference*, pages 253–259, 1992.

[Gla90] R.J. van Glabbeek. *Comparative Concurrency Semantics and Refinement of Actions*. PhD thesis, Free University, Amsterdam, 1990. Second edition available as *CWI tract* 109, CWI, Amsterdam 1996.

[Gla93] R.J. van Glabbeek. The linear time – branching time spectrum II; the semantics of sequential systems with silent moves (extended abstract). In E. Best, editor, Proceedings *CONCUR'93,* 4th International Conference on Concurrency Theory, Hildesheim, Germany, August 1993, volume 715 of *Lecture Notes in Computer Science*, pages 66–81. Springer-Verlag, 1993.

[Gon95] Michael Gondon. The semantic challenge of Verilog HDL. In *the Tenth Annual IEEE Symposium on Logic in Computer Science (LICS'95)*, 1995.

[Gop92] Ganesh Gopalakrishnan. Asynchronous circuit verification using trace theory and CCS. Technical Report UUCS-92-008, Computer Science Department, University of Utah, 1992.

[Gor87] Michael J. C. Gordon. HOL: A proof generating system for Higher-Order Logic. Technical Report 103, University of Cambridge, UK, 1987.

[HB97] Henrik Hulgaard and Steven M. Burns. Bounded delay timing analysis of a class of CSP programs. *Formal Methods in System Design*, 11(3):265–294, October 1997.

[Her97] Christian Hernalsteen. A timed automaton model for ET-LOTOS verification. In Tadanori Mizuno, Norio Shiratori, Teruo Higashino, and Atsushi Togashi, editors, *Proc. Formal Description Techniques X/Protocol Specification, Testing and Verification XVII*, pages 193–204. Chapman-Hall, London, UK, November 1997.

[HG92] C. Anthony R. Hoare and Michael J. C. Gordon, editors. *Mechanized Reasoning and Hardware Design*. International Series in Computer Science. Prentice Hall, UK, 1992.

[HHK96] R. H. Hardin, Z. HarEl, and R. P. Kurshan. COSPAN. In Rajeev Alur and Thomas A. Henzinger, editors, *Computer Aided Verification '96*, volume 1102 of *Lecture Notes in Computer Science*, pages 423–427. Springer-Verlag, Berlin, Germany, 1996.

[HM80] Matthew Hennessy and A. J. Robin G. Milner. *On Observing Nondeterminism and Concurrency*, volume 85 of *Lecture Notes in Computer Science*, pages 299–309. 1980.

[Hoa85] C. Anthony R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, Englewood Cliffs, New Jersey, USA, 1985.

[Hol91] Gerard J. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall, Englewood Cliffs, New Jersey, USA, 1991.

[HYHD95] R. C. Ho, C. H. Yang, Mark A. Horowitz, and David L. Dill. Architecture validation for processors. In *Proc. 22nd. Annual International Synposium on Computer Architecture*, 1995.

[HYKT94] Teruo Higashino, Keiichi Yasumoto, Junji Kitamichi, and Kenichi Taniguchi. Hardware synthesis from a restricted class of LOTOS expressions. In Son Vuong, editor, *Proc. Protocol Specification, Testing and Verification XIV*. Chapman-Hall, London, UK, 1994.

[IEE93] IEEE. *VHSIC Hardware Design Language*. IEEE 1076. Institution of Electrical and Electronic Engineers Press, New York, USA, 1993.

[IEE95] IEEE. *IEEE Standard Hardware Design Language based on the Verilog Hardware Description Language*. IEEE 1364. Institution of Electrical and Electronic Engineers Press, New York, USA, 1995.

[IEE99] IEEE. Draft standard for standard delay format (SDF) for the electronic design process. Technical report, May 1999.

[ISO89] ISO/IEC. *Information Processing Systems – Open Systems Interconnection –* LOTOS *– A Formal Description Technique based on the Temporal Ordering of Observational Behaviour*. ISO/IEC 8807. International Organization for Standardization, Geneva, Switzerland, 1989.

[ISO91] ISO/IEC. *Information Processing Systems – Open Systems Interconnection – Conformance Testing Methodology and Framework – Part 3: The Tree and Tabular Combined Notation (TTCN)*. ISO/IEC 9646-3. International Organization for Standardization, Geneva, Switzerland, 1991.

[ISO94] ISO/IEC. *Information Processing Systems – Open Systems Interconnection – Basic Reference Model*. ISO/IEC 7498. International Organization for Standardization, Geneva, Switzerland, 1994.

[ISO95] ISO/IEC. *Information Processing Systems – Open Distributed Processing – Basic Reference Model*. ISO/IEC 10746. International Organization for Standardization, Geneva, Switzerland, 1995.

[ISO97] ISO/IEC. *Information Technology – Open Systems Interconnection –* ESTELLE*: A Formal Description Technique based on an Extended State Transition Model*. ISO/IEC 9074. International Organization for Standardization, Geneva, Switzerland, 1997.

[ISO98] ISO/IEC. *Information Processing Systems – Open Systems Interconnection – Enhancements to* LOTOS *– A Formal Description Technique based on the Temporal Ordering of Observational Behaviour*. ISO/IEC CD. International Organization for Standardization, Geneva, Switzerland, April 1998.

[ITU92] ITU. *Guidelines for the Application of ESTELLE, LOTOS and SDL*. International Telecommunications Union, Geneva, Switzerland, October 1992.

[ITU95] ITU. *SDL combined with ASN.1*. ITU-T Z.105. International Telecommunications Union, Geneva, Switzerland, 1995.

[JS90] Geraint Jones and Mary Sheeran. Circuit design in Ruby. In Jørgen Staunstrup, editor, *Formal Methods For VLSI Design*, pages pp.13–70. Elsevier Science Publishers, Amsterdam, Netherlands, 1990.

[JT97] Ji He and Kenneth J. Turner. Extended DILL: Digital logic with LOTOS. Technical Report CSM-142, Department of Computing Science and Mathematics, University of Stirling, UK, November 1997.

[JT99a] Ji He and Kenneth J. Turner. Protocol-inspired hardware testing. In Gyula Csopaki, Sarolta Dibuz, and Katalin Tarnay, editors, *Proc. Testing Communicating Systems XII*, pages 131–147, London, UK, September 1999. Kluwer Academic Publishers.

[JT99b] Ji He and Kenneth J. Turner. Specification and verification of synchronous hardware using LOTOS. In Jianping Wu, Samuel T. Chanson, and Quiang Gao, editors, *Proc. Formal Methods for Protocol Engineering and Distributed Systems (FORTE XII/PSTV XIX)*, pages 295–312, London, UK, October 1999. Kluwer Academic Publishers.

[JU93] M. B. Josephs and J. T. Udding. An overview of DI algebra. In T. N. Mudge, V. Milutinovic, and L. Hunter, editors, *Proc. Hawaii International Conf. System Sciences*, volume I, pages 329–338. IEEE Computer Society Press, January 1993.

[KB95] C. Delgado Kloos and Peter T. Breuer, editors. *Semantics of VHDL*. Kluwer Academic Publishers, London, UK, 1995.

[Kir94] Carron E. Kirkwod. *Verification of LOTOS Specifications using Term Rewriting Techniques*. PhD thesis, University of Glasgow, 1994.

[KKTT98] Alex Kondratyev, Michael Kishinevsky, Alexander Taubin, and Sergei Ten. Analysis of Petri nets by ordering relations in reduced unfoldings. *Formal Methods in System Design*, 12(1):5–38, January 1998.

[KKTV94] M. Kishinevsky, A. Kondratyev, A. Taubin, and V. Varshavsky. Analysis and identification of speed-independent circuits on an event model. *Formal Methods in System Design*, 4(1):33–75, 1994.

[Kli83] Raymond Kline. *Structured Digital Design, Including MSI/LSI Components and Microprocessors*. Prentice-Hall, Englewood Cliffs, New Jersey, USA, 1983.

[L9́7] Luc Léonard. *A Timed Extension of LOTOS*. PhD thesis, Universite de Liege, 1997.

[Lar90] K. G. Larsen. Proof systems for satisfiability in hennessy-milner logic with recursion. *Theoretical Comupter Science*, 72:256–288, 1990.

[Led92] G. Leduc. A framework based on implementation relations for implementing LOTOS specifications. *Computer Networks and ISDN Systems*, 25(1):23–41, August 1992.

[LL94] Luc Léonard and Guy Leduc. An enhanced version of timed LOTOS and its application to a case study. In Richard L. Tenney, Paul D. Amer, and M. Ümit Uyar, editors, *Proc. Formal Description Techniques VI*, pages 483–500. North-Holland, Amsterdam, Netherlands, 1994.

[LL97] Luc Léonard and Guy Leduc. An introduction to ET-LOTOS for the description of time-sensitive systems. *Computer Networks and ISDN Systems*, 29(2):271–292, February 1997.

[LR95] Luis Llana and Gualberto Rabay Filho. Defining equivalences between time/action graphs and timed action graphs. Technical report, Department of Telematic Systems Engineering, Polytechnic University of Madrid, Spain, December 1995.

[MAH98] D. Moundanos, A. Abraham, and Y. V. Hoskote. Abstraction techniques for validation coverage analysis and test generation. *IEEE Transactions on Computers*, 47:2–14, 1998.

[Mar90] Alain J. Martin. The limitations to delay-insensitivity in asynchronous circuits. In William J. Dally, editor, *Advanced Research in VLSI*, pages 263–278. MIT Press, 1990.

[MB59] David E. Muller and W. S. Bartky. A theory of asynchronous circuits. In *Proceedings of an International Symposium on the Theory of Switching*, pages 204–243. Harvard University Press, April 1959.

[MC93] J.D. Morison and A.S. Clarke. *ELLA 2000: A Language for Electronic System Design*. McGraw-Hill, New York, USA, 1993.

[McM93] Kenneth L. McMillan. *Symbolic Model Checking*. Kluwer Academic, 1993.

[Mel93] T. Melham. *Higher Order Logic and Hardware Verification*. Number 31 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1993.

[MFR85] Charles E. Molnar, Ting-Pien Fang, and Frederick U. Rosenberger. Synthesis of delay-insensitive modules. In Henry Fuchs, editor, *1985 Chapel Hill Conference on Very Large Scale Integration*, pages 67–86. Computer Science Press, 1985.

[MGG93] T.F. Melham, N. J. C. Gordon, and M. J. Gordon, editors. *Introduction to Hol : A Theorem Proving Environment for Higher Order Logic*. Cambridge University Press, 1993.

[Mil89] A. J. Robin G. Milner. *Communication and Concurrency*. Addison-Wesley, Reading, Massachusetts, USA, 1989.

[Mil95] George J. Milne. Modeling digital hardware in a process algebra. Technical Report CIS-95-010, University of South Australia, School of Computer and Information Science, May 1995.

[MM92] George A. McCAskill and George J. Milne. Hardware description and verification using the CIRCAL system. Technical Report HDV-24-92, Department of Computer Science, University of Strathclyde, Glasgow, UK, June 1992.

[MM93] George A. McCaskill and George J. Milne. Sequential circuit analysis with a BDD based process algebra system. Technical Report HDV-25-93, Department of Computer Science, University of Strathclyde, January 1993.

[Mou94] Laurent Mounie. A lotos specification of a ”Transit-Node”. Technical Report SPECTRE 94-8, VERIMAG, Grenoble, France, March 1994.

[MP95] O. Maler and A. Pnueli. Timing analysis of asynchronous circuits using timed automata. In P.E. Camurati and H. Eveking, editors, *CHARME'95*, volume 987 of *Lecture Notes in Computer Science*, pages 189–205. Springer-Verlag, 1995.

[MT94] U. Martin and M. Thomas. Verification techniques for lotos. In *Industrial Benefit of Formal Methods, FME'94*, volume 873 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994.

[MY96] O. Maler and S. Yovine. Hardware timing verification using kronos. In *7th Conference on Computer-based Systems and Software Engineering*. IEEE Press, 1996.

[Nic87] R. De Nicola. Extensional equivalences for transition systems. *Acta Informatica*, 24:211–237, 1987.

[NUK+94] T. Nanya, Y. Ueno, H. Kagotani, M. Kuwako, and A. Takamura. Titac design of a quasi-delay-insensitive microprocessor. *IEEE Design and Test of Computers*, 11(2):50–63, 1994.

[O'D95] John O'Donnell. From transistors to computer architecture: Teaching functional circuit specification in hydra. In *FPLE'95: Symposium on Functional Programming Language in Education*, volume 1022 of *Lecture Notes in Computer Science*, pages 195–214, 1995.

[PF90] D. H. Pitt and D. Freestone. The derivation of conformance tests from LOTOS specifications. *IEEE Transactions on Software Engineering*, 16(12):1337–1343, December 1990.

[PLR95] Santiago Pavón, David Larrabeiti, and Gualberto Rabay Filho. LOLA user manual (version 3R6). Technical report, Department of Telematic Systems Engineering, Polytechnic University of Madrid, Spain, February 1995.

[Pnu77] Amir Pnueli. The temporal logic of programs. In *Proc. 19th. IEEE Annual Symposium on Foundations of Computer Science*, pages 46–57, Providence, USA, November 1977.

[QPF89] Juan Quemada, Santiago Pavon, and Angel Fernández. Transforming LOTOS specifications with LOLA: The parameterized expansion. In Kenneth J. Turner, editor, *Proc. Formal Description Techniques I*, pages 45–54. North-Holland, Amsterdam, Netherlands, 1989.

[Rei85] W. Reisig. *Petri-Nets: An Introduction*. EATCS Monographs on Theoretical Computer Science. Springer-Verlag, Berlin, Germany, 1985.

[RQ96] Gualberto Rabay Filho and Juan Quemada. TE-LOLA: A timed LOLA prototype. In Zmago Brezocnik and Tatjana Kapus, editors, *Proc. COST 247 International Workshop on Applied Formal Methods*, pages 85–95, Slovenia, June 1996. University of Maribor.

[RSM97] J. M. T. Romijn, O. Sies, and J. R. Moonen. A two-level approach to automated conformance testing of VHDL designs. *Testing of Communicating Systems*, 10:432–447, 1997.

[SK96] Jørgen Staunstrup and Thomas Kropf. IFIP WG10.5 benchmark circuits. http: //goethe. ira. uka.de/hvg/benchmarks.html, July 1996.

[SLM+96] Luis Sánchez Fernández, Marin L. López, Natividad Mart́nez Madrid, C. Carreras, J. C. López, Carlos Delgado Kloos, A. Royo, and Peter T. Breuer. Co-design at work: The Ethernet bridge case study. *Current Issues in Electronic Modelling*, 8, April 1996.

[SRC96] Mandayam Srivas, Harald Ruess, and David Cyrluk. Hardware verification using PVS. In Thomas Kropf, editor, *Formal Hardware Verification Methods and Systems in Comparison*, volume 1287 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.

[SRI91] SRI. *The HOL System: Description, Tutorial, Reference Manual*. SRI International Cambridge Research Center and TSTO Australia, 1991.

[SSM94] R. F. Sproull, I. E. Sutherland, and C. E. Molnar. The counterflow pipeline processor architecture. *IEEE Design and Test of Computers*, 11(3):48–59, 1994.

[ST97] Carron Shankland and Muffy Thomas. Symbolic bisimulations for full lotos. In *Algebraic Methodology and Software Technology'97*, volume 1349 of *Lecture Notes in Computer Science*, pages 479–493. Springer-Verlag, 1997.

[Sta93] J. Staunstrup. IFIP WG 10.2 collection of circuit verification examples, 1993.

[Sta97] J. Staunstrup. Design verification using synchronized transitions. In *Formal Hardware Verification. Methods and Systems in Comparison*. Springer-Verlag, Berlin, Germany, 1997.

[Sut89] Ivan E. Sutherland. Micropipelines. *Communications of the ACM*, 32(6):720–738, June 1989.

[Tre96] J. Tretmans. Test generation with inputs, outputs and repetitive quiescence. *Software Concepts and Tools*, 17:103–120, 1996.

[TS94] Kenneth J. Turner and Richard O. Sinnott. DILL: Specifying digital logic in LOTOS. In Richard L. Tenney, Paul D. Amer, and M. Ümit Uyar, editors, *Proc. Formal Description Techniques VI*, pages 71–86. North-Holland, Amsterdam, Netherlands, 1994.

[TTW97] G. K. Theodoropoulos, G. K. Tsakogiannis, and J. V. Woods. Occam: an asynchronous hardware description language? In *Proceedings of the 23rd EUROMICRO Conference: New Frontiers of Information Technology*, pages 249–256, September 1997.

[Tur93] Kenneth J. Turner, editor. *Using Formal Description Techniques — An Introduction to ESTELLE, LOTOS and SDL*. Wiley, New York, January 1993.

[Tur94] Kenneth J. Turner. Exploiting the *m4* macro language. Technical Report CSM-126, Department of Computing Science and Mathematics, University of Stirling, UK, September 1994.

[Udd86] Jan Tijmen Udding. A formal model for defining and classifying delay-insensitive circuits. *Distributed Computing*, 1(4):197–204, 1986.

[Ung69] Stephen H. Unger. *Asynchronous Sequential Switching Circuits*. Wiley-Interscience, UK, 1969.

[van91] Peter H. J. van Eijk. The LOTOSPHERE integrated tool environment LITE. In Kenneth R. Parker and Gordon A. Rose, editors, *Proc. Formal Description Techniques IV*, pages 471–474. North-Holland, Amsterdam, Netherlands, November 1991.

[VK95] F. Vemuri and R. Kalyanaraman. Generation of design verification tests from behavioral VHDL programs using path enumeration and constraint programming. *IEEE Transactions on Very Large Scale Integration Systems*, 3:201–214, 1995.

[vW89] R. J. van Glabbeek and W. P. Weijland. Branching time and abstraction in bisimulation. Technical Report CS R8911, Centrum voor Wiskunde en Informatica, Amsterdam, 1989.

[Wan91] Yi Wang. CCS + time=An interleaving model for real-time systems. In *Proc. 18th International Colloquium on Automata Languages and Programming*, number 510 in Lecture Notes in Computer Science, pages 217–228. Springer-Verlag, Berlin, Germany, 1991.

[Wez89] Clazien D. Wezeman. The CO-OP method for compositional derivation of conformance testers. In Ed Brinksma, Giuseppe Scollo, and Christopher A. Vissers, editors, *Proc. Protocol Specification, Testing and Verification VIII*. North-Holland, Amsterdam, Netherlands, June 1989.

[WP80] D. Winkel and F. Prosser. *The Art of Digital Design*. Prentice-Hall, Englewood Cliffs, New Jersey, USA, 1980.

[YK98] A. V. Yakovlev and A. M. Koelmans. Petri nets and digital hardware design. In *Lectures on Petri Nets II: Applications. Advances in Petri Nets*, volume 1492 of *Lecture Notes in Computer Science*, pages 154–236, 1998.

A Glossary

ACTL: Action based Computational Tree Logic [DV90], a temporal logic which is similar to CTL but is interpreted over actions.

ADT: Abstract Data Type, a language which support ADT does not imply the particular implementations of data types.

Asynchronous Circuit: A circuit in which components change state independently at their own rates.

Behavioural Specification: A behavioural specification looks at a system as a black box. It specifies the behaviour of a circuit exhibited on its interface to the environment.

BDD: Binary Decision Diagram, a data structure for representing a boolean function [Bry92].

Basic Logic Gate: a basic component that evaluates a simple logical function such as *and*, *nand* (not and), *xor* (exclusive or).

Bounded Delay: a component has bounded delay if an upper and lower bound for the delay magnitude is known.

CCS: Calculus of Communicating Systems [Mil89], a process algebra used to specify parallel and concurrent systems.

CIRCAL: Circuit Calculus [MM92]. A process algebra derived from CCS for specifying and analysing digital circuits.

Combinational Circuits: Circuits whose outputs depend only on the current inputs.

CSP: Communicating Sequential Processes [Hoa85], a process algebra used to specify parallel and concurrent systems.

CTL: Computational Tree Logic [CES86], a branching time temporal logic which is interpreted over states.

Design Verification: One of the hardware verification tasks which checks if an implementation of a circuit design satisfies some properties.

DI Circuit: A delay insensitive circuit assumes unbounded delays on its wires and components, thus the correct function of the circuit is insensitive to the actual delays on wires and components.

E-LOTOS: Enhancements to LOTOS [ISO98].

ELLA: A hardware design language from DRA Malvern.

ET-LOTOS: Enhanced Timed-LOTOS [LL97].

Flip-Flop: Clocked one-bit memory element whose output is decoupled from its input. New data may be read into a flip-flop while previous data is being output. A D (Delay) flip-flop has a single data input that is read on clock signals. A JK flip-flop has two data inputs (corresponding to 0 and 1 outputs) that are read on clock signals. Other varieties include MS (Master-Slave), RS (Reset-Set) and T (Trigger) flip-flops.

Fundamental Mode: The environment of a circuit is said to be in fundamental mode if it can provides inputs only when the circuit is stable.

Hazard: Transient and undesired signal transitions appeared on the outputs of digital circuits in response to the changes on inputs.

HDL: A hardware description language is a language for rigorous definition of hardware components and circuits. It generally provides support for multi-level description and realisation of hardware, and may have a formal basis.

HML: Hennessy-Milner Logic [HM80], a logic which is interpreted over actions and is used to express the properties of concurrent systems.

HOL: Higher Order Logic. A proof generating system for higher order logic [Gor87].

Inertial Delay: An inertial delay component may filter out the narrow pulses of its input signals, consequently the shape of its output waveform may be altered.

Input Receptive: The specification of a circuit is input receptive if every input is acceptable at every state of the specification.

Input/Output Mode: The environment of a circuit is said to be in input/output mode if it is allowed to provide inputs no matter if the circuit is stable or not.

Implementation Verification: One of the hardware verification tasks which checks if a circuit specification related to another one with respect to a formal relation.

LOTOS: Language of Temporal Ordering Specification [ISO89], a formal language standardised by ISO in 1989.

LTS: A Labelled transition system is an automaton consisting of a set of states (including the initial state), a set of actions and a set of transitions. Each transition is related to two states and an action, representing that the system changes its state from one to the other after the action takes place.

Model Checking: The method of formally verifying whether a finite-state model satisfies some properties.

μ-Calculus: A modal logic [Lar90] which is an extension of HML.

Pure Delay: A pure delay component does not change the waveform of its input signal; all the signal transitions are simply delay by a certain magnitude.

QDI: A quasi-delay insensitive circuit assumes unbounded delays on its wires and components, but delays on forked wires are assumed to be the same.

Occam: A language based on CSP developed in INMOS to specify concurrent processes which communicate via one-way channels.

Race Condition: a situation where the relative speeds of components decides the behaviour of a circuit. A race condition is usually undesirable as it can lead to non-determinism of digital circuits.

Requirements Capture: One of the hardware verification tasks which checks if a circuit specification is what it should be.

RTL: A specification at register transfer level specifies the data flows between registers of digital circuits.

Ruby: A relational language for describing and designing circuits [JS90].

Semi-Modular: A circuit is semi-modular if for all components in the circuit, their inputs cannot change any pending outputs.

Sequential Circuit: A circuit whose outputs depend on the states of components at a previous time. Sequential circuits generally have some kind of feedback, such that previous outputs affect future values.

SI Circuit: A speed independent circuit assume zero delays on its wires but unbounded delays on its components. The correct function of the circuit is independent the actual delays on components.

Structural Specification: A structural specification specifies how a system is built from smaller components.

Synchronous Circuits: A circuit in which components change state under the control of a master clock.

Unbounded Delay: A component has unbounded delay if the upper bound of its delay magnitude is unknown, except that it is positive and finite.

TE-LOTOS: Time Extended LOTOS [RQ96].

Temporal Logic: A logic with the notion of time involved. A temporal logic formula can express the fact about past, present and future.

VHDL: Very High Speed Integrated Circuit (VHSIC) Hardware Description Language. An IEEE standard HDL [IEE93].

Verilog: A hardware description language which was standardised by IEEE [IEE95].

B DILL Library Components

This section summarises the components in the DILL library. The components of synchronous circuits are found in table 7, of asynchronous circuits in table 8, and of timed circuits in table 9.

Component	**Variants**
Adder	2/4 inputs, behavioural/structural, half/full/parallel/ripple
And, ...	2/3/4/8 inputs, 0/1-active
Clock	-
Comparator	1/4/8/*n* inputs, behavioural/structural
Counter	behavioural/structural
Decoder	2/3 inputs, behavioural/structural, 0/1-active outputs, BCD-Decimal/Excess-3/Gray
Demultiplexer	1/2 inputs, behavioural/structural
Divider	2/4/8 inputs, behavioural/structural, +ve/-ve trigger
Encoder	4/8 inputs, behavioural/structural, 0/1-active outputs
FlipFlop	D/JK/MS/RS/T, behavioural/structural, +ve/-ve trigger, preset, preclear, lockout
Inverter	1/4/8 inputs, 0/1-active tri-state enable
Latch	D/RS, 1/4/8 bits, behavioural/structural, preset, preclear, clocked
Memory	behavioural/structural
Multiplexer	2/4 inputs, 1/8/*n*-bit, behavioural/structural
One, ...	source of logic 1/0, sink
Parity	8 inputs, white-box/gate-level
Register	4/8/*n* bits, black-box/gate-level, +ve/-ve trigger, load enable/preclear, bucket brigade/pass-on/shift
Repeater	1/4/8 inputs, 0/1-active

Table 7: The components of synchronous circuits in the DILL library

Component	**Function**
And, ...	basic logic gates
C_Element (Join)	used for synchronising signal transitions
Fork	forking wires
Latch	storage components
Merge	for merging signal transitions on two inputs
RGD Arbiter	request–grant–done arbiter
Selector	selecting nondeterminisiticly from two inputs
Sequencer	sequencing two inputs
Wire	for explicitly introducing delay

Table 8: The components of asynchronous circuits

LOTOS Syntax

This section gives selected syntax for LOTOS in table 10. Only the informal explanations are presented here. The definition of LOTOS syntax and its formal semantics can be found in [ISO89].

ET-LOTOS Syntax

This section gives the syntax of ET-LOTOS which is related to its timing featrues. The definition of the syntax of ET-LOTOS and its formal semantics can be found in [LL97].

Component	Function
GeneralDelay	general delays, can include inertial and pure delays
Hold	hold timing constraints
InertialDelay	inertial delays
Period	timing constraints, for checking the periods of signals
PureDelay	pure delays
Setup	setup timing constraint
Width	timing constraints, for checking the widths of signals

Table 9: Timed components in the DILL library

Notation	**Meaning**
(* text *)	a comment
stop	a behaviour that does nothing (no further action)
exit	a behaviour that immediately terminates successfully
exit (results)	successful termination with result values
gate	a `port' at which event offers may synchronise
gate !value	an offer to synchronise on a given value
gate ?variable:sort	an offer to synchronise on any value of the given sort, binding the actual value to the given variable name
gate !... ?... [predicate]	an event offer with a predicate on values synchronised
process name [gates] (parameters) : **noexit** := behaviour	a named process abstraction with given gates and value parameters, but no termination (e.g. it repeats indefinitely)
process name [gates] (parameters) : **exit** (results) := behaviour	a process that terminates successfully with the given result sorts
name [gates] (parameters)	an instantiation of a named process
offer ; behaviour	prefixes an event offer to some behaviour (`followed by')
[guard] $\rightarrow$ behaviour	offers behaviour only if the guard condition is satisfied (`if')
behaviour1 [] behaviour2	offers a choice between two behaviours (`or')
behaviour1 $\gg$ behaviour2	allows the second behaviour to occur if the first behaviour terminates successfully (`enables')
exit (results) $\gg$ **accept** declarations **in** behaviour	successful termination with export of result values
behaviour1 $\rhd$ behaviour2	allows the second behaviour to disrupt the first behaviour unless this terminates successfully first (`disabled by')
behaviour1 || behaviour2	allows two behaviours to run in parallel, but fully synchronised on their events (`synchronised with')
behaviour1 ||| behaviour2	allows two behaviours to run in parallel, but with independent occurrence of their events (`interleaved with')
behaviour1 |[gates]| behaviour2	allows two behaviours to run in parallel, synchronising on all events at the given gates (`synchronised on *gates* with')

Table 10: Selected LOTOS syntax

Notation	**Meaning**
Δ^d Q	process *Q* is delayed by *d*
exit {d}	successful termination within [0, d], otherwise behaves like **stop**
exit (results) {d}	same as above but termination with results
gate !... ?... @ t	the time when *gate !... ?...* takes place is recorded in *t*
gate !... ?... {d}	*gate !... ?...* happens within [0, d], otherwise behaves like **stop**
gate !... ?... @t [f(t)]	the time when the event takes place satisfies *f(t)*, otherwise **stop**
i {d}	**i** must happen within [0, d]
i @ t {d}	**i** mut happen within [0, d] and the time is recorded in *t*

Table 11: Seclected ET-LOTOS syntax