FORMAL VERIFICATION OF ASYNCHRONOUS SYSTEMS

by

BUDI RAHARDJO

A Thesis
Submitted to the Faculty of Graduate Studies
in Partial Fulfillment of the Requirements
for the Degree of

Doctor of Philosophy

Department of Electrical and Computer Engineering
University of Manitoba
Winnipeg, Manitoba

ISBN 0-612-16239-7

Name _____

Dissertation Abstracts International and *Masters Abstracts International* are arranged by broad, general subject categories. Please select the one subject which most nearly describes the content of your dissertation or thesis. Enter the corresponding four-digit code in the spaces provided.

ELECTRONICS AND ELECTRICAL
SUBJECT TERM

0544
SUBJECT CODE

UMI

Subject Categories

THE HUMANITIES AND SOCIAL SCIENCES

COMMUNICATIONS AND THE ARTS
Architecture 0729
Art History 0377
Cinema 0900
Dance 0378
Fine Arts 0357
Information Science 0723
Journalism 0391
Library Science 0399
Mass Communications 0708
Music 0413
Speech Communication 0459
Theater 0465

EDUCATION
General 0515
Administration 0514
Adult and Continuing 0516
Agricultural 0517
Art 0273
Bilingual and Multicultural 0282
Business 0688
Community College 0275
Curriculum and Instruction 0727
Early Childhood 0518
Elementary 0524
Finance 0277
Guidance and Counseling 0519
Health 0680
Higher 0745
History of 0520
Home Economics 0278
Industrial 0521
Language and Literature 0279
Mathematics 0280
Music 0522
Philosophy of 0998
Physical 0523
Psychology 0525
Reading 0535
Religious 0527
Sciences 0714
Secondary 0533
Social Sciences 0534
Sociology of 0340
Special 0529
Teacher Training 0530
Technology 0710
Tests and Measurements 0288
Vocational 0747

LANGUAGE, LITERATURE AND LINGUISTICS
Language
- General 0679
- Ancient 0289
- Linguistics 0290
- Modern 0291

Literature
- General 0401
- Classical 0294
- Comparative 0295
- Medieval 0297
- Modern 0298
- African 0316
- American 0591
- Asian 0305
- Canadian (English) 0352
- Canadian (French) 0355
- English 0593
- Germanic 0311
- Latin American 0312
- Middle Eastern 0315
- Romance 0313
- Slavic and East European 0314

PHILOSOPHY, RELIGION AND THEOLOGY
Philosophy 0422
Religion
- General 0318
- Biblical Studies 0321
- Clergy 0319
- History of 0320
- Philosophy of 0322

Theology 0469

SOCIAL SCIENCES
American Studies 0323
Anthropology
- Archaeology 0324
- Cultural 0326
- Physical 0327

Business Administration
- General 0310
- Accounting 0272
- Banking 0770
- Management 0454
- Marketing 0338

Canadian Studies 0385
Economics
- General 0501
- Agricultural 0503
- Commerce-Business 0505
- Finance 0508
- History 0509
- Labor 0510
- Theory 0511

Folklore 0358
Geography 0366
Gerontology 0351
History
- General 0578
- Ancient 0579
- Medieval 0581
- Modern 0582
- Black 0328
- African 0331
- Asia, Australia and Oceania 0332
- Canadian 0334
- European 0335
- Latin American 0336
- Middle Eastern 0333
- United States 0337

History of Science 0585
Law 0398
Political Science
- General 0615
- International Law and Relations 0616
- Public Administration 0617

Recreation 0814
Social Work 0452
Sociology
- General 0626
- Criminology and Penology 0627
- Demography 0938
- Ethnic and Racial Studies 0631
- Individual and Family Studies 0628
- Industrial and Labor Relations 0629
- Public and Social Welfare 0630
- Social Structure and Development 0700
- Theory and Methods 0344

Transportation 0709
Urban and Regional Planning 0999
Women's Studies 0453

THE SCIENCES AND ENGINEERING

BIOLOGICAL SCIENCES
Agriculture
- General 0473
- Agronomy 0285
- Animal Culture and Nutrition 0475
- Animal Pathology 0476
- Food Science and Technology 0359
- Forestry and Wildlife 0478
- Plant Culture 0479
- Plant Pathology 0480
- Plant Physiology 0817
- Range Management 0777
- Wood Technology 0746

Biology
- General 0306
- Anatomy 0287
- Biostatistics 0308
- Botany 0309
- Cell 0379
- Ecology 0329
- Entomology 0353
- Genetics 0369
- Limnology 0793
- Microbiology 0410
- Molecular 0307
- Neuroscience 0317
- Oceanography 0416
- Physiology 0433
- Radiation 0821
- Veterinary Science 0778
- Zoology 0472

Biophysics
- General 0786
- Medical 0760

EARTH SCIENCES
Biogeochemistry 0425
Geochemistry 0996
Geodesy 0370
Geology 0372
Geophysics 0373
Hydrology 0388
Mineralogy 0411
Paleobotany 0345
Paleoecology 0426
Paleontology 0418
Paleozoology 0985
Palynology 0427
Physical Geography 0368
Physical Oceanography 0415

HEALTH AND ENVIRONMENTAL SCIENCES
Environmental Sciences 0768
Health Sciences
- General 0566
- Audiology 0300
- Chemotherapy 0992
- Dentistry 0567
- Education 0350
- Hospital Management 0769
- Human Development 0758
- Immunology 0982
- Medicine and Surgery 0564
- Mental Health 0347
- Nursing 0569
- Nutrition 0570
- Obstetrics and Gynecology 0380
- Occupational Health and Therapy 0354
- Ophthalmology 0381
- Pathology 0571
- Pharmacology 0419
- Pharmacy 0572
- Physical Therapy 0382
- Public Health 0573
- Radiology 0574
- Recreation 0575
- Speech Pathology 0460
- Toxicology 0383

Home Economics 0386

PHYSICAL SCIENCES
Pure Sciences
Chemistry
- General 0485
- Agricultural 0749
- Analytical 0486
- Biochemistry 0487
- Inorganic 0488
- Nuclear 0738
- Organic 0490
- Pharmaceutical 0491
- Physical 0494
- Polymer 0495
- Radiation 0754

Mathematics 0405
Physics
- General 0605
- Acoustics 0986
- Astronomy and Astrophysics 0606
- Atmospheric Science 0608
- Atomic 0748
- Electronics and Electricity 0607
- Elementary Particles and High Energy 0798
- Fluid and Plasma 0759
- Molecular 0609
- Nuclear 0610
- Optics 0752
- Radiation 0756
- Solid State 0611

Statistics 0463

Applied Sciences
Applied Mechanics 0346
Computer Science 0984
Engineering
- General 0537
- Aerospace 0538
- Agricultural 0539
- Automotive 0540
- Biomedical 0541
- Chemical 0542
- Civil 0543
- Electronics and Electrical 0544
- Heat and Thermodynamics 0348
- Hydraulic 0545
- Industrial 0546
- Marine 0547
- Materials Science 0794
- Mechanical 0548
- Metallurgy 0743
- Mining 0551
- Nuclear 0552
- Packaging 0549
- Petroleum 0765
- Sanitary and Municipal 0554
- System Science 0790

Geotechnology 0428
Operations Research 0796
Plastics Technology 0795
Textile Technology 0994

PSYCHOLOGY
General 0621
Behavioral 0384
Clinical 0622
Developmental 0620
Experimental 0623
Industrial 0624
Personality 0625
Physiological 0989
Psychobiology 0349
Psychometrics 0632
Social 0451

THE UNIVERSITY OF MANITOBA

FACULTY OF GRADUATE STUDIES

COPYRIGHT PERMISSION

FORMAL VERIFICATION OF ASYNCHRONOUS SYSTEMS

BY

BUDI RAHARDJO

A Thesis/Practicum submitted to the Faculty of Graduate Studies of the University of Manitoba in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Acknowledgments

First, I would like to thank my advisor, Prof. R. D. McLeod, without whom I probably would not have done my Ph.D. I admire his leadership and thank him for his support. I am thankful that he has been my advisor.

I appreciate the discussions, friendship, and support from everyone in the Electrical and Computer Engineering Department (especially in the VLSI group); Chinsong Sul for introducing me to asynchronous circuits, Dave Blight for reading and correcting my English in numerous papers and articles, Tapas Shome for numerous discussions on formal methods and engineering in general, Wishnu Prasetya of Eindhoven University for giving me a HOL tutorial over the Internet through e-mails (here's a real example of the use of Internet), Armein Langi, Ken Ferens, Zaifu Zhang for their friendship, Prof. Howard Card for his comments and questions (the way he thinks sometimes amazes me), Yair Bourlas for many SDL discussions, and others. I would also like to thank my colleagues at "PAU Mikroelektronika" in Indonesia for their support, Bill Reid and Kathy Norman of Computer Services University of Manitoba for providing a place for part-time work, and *Telecommunication and Research Laboratories* (TRLabs) for their facilities and support. I appreciate the support and patience from my parents in Indonesia.

Most of all, I would like to thank my wife (Dian) and children (Atika and Luqman) for their enormous patience and love. This thesis is for you.

Abstract

Asynchronous systems have several advantages as compared with synchronous systems. The advantages, due to their local communications, include faster speed, modularity, and lower power. Unfortunately, asynchronous systems are difficult to design and verify. This thesis is one of the first attempts to identify system level design methodologies specifically oriented to asynchronous systems, with emphasis on the use of formal verification methods as an integral part of the methodology. This identification is accomplished by considering various levels of abstraction from asynchronous circuits to asynchronous systems.

As opposed to reinventing tools, this thesis adapted existing tools for use in designing and verifying asynchronous systems. Specific instances include: the use of SPIN to verify asynchronous circuits modeled in PROMELA, and the use of SDT to validate and verify asynchronous systems modeled in SDL. In the former, hazardous circuits (in input-output mode and fundamental mode) were modeled in PROMELA, and SPIN was used to show the presence of hazards. In the latter, a Counterflow Pipeline Processor modeled in SDL and SDT was used to validate and verify the design. Initial work on proof-based verification of asynchronous circuits is also presented by modeling asynchronous circuits in *Real-Time CSP* (RTCSP) and higher-order logic.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

There are two key phrases in the title of this thesis, "*formal verification*" and "*asynchronous systems*". These are the topics of study in this thesis: the use of formal techniques to verify that an asynchronous system satisfies its specification. This thesis attempts to identify system level design methodologies specifically oriented to asynchronous systems with emphasis on the use of formal methods as an integral part of the methodology.

Asynchronous systems are systems that operate without a global synchronization signal. In the context of this thesis, an asynchronous system is typically a hardware and/or software design. In terms of hardware design, asynchronous circuits are digital circuits that operate without a global clock. This type of circuit has numerous advantages as compared with conventional synchronously-clocked circuits. Unfortunately, correct asynchronous circuits are difficult to design, verify, and validate. Hazards and critical races are notoriously difficult to manage. This topic is elaborated further in Chapter 2, "Overview of Design Methodologies for Asynchronous Circuits."

Formal verification, the main focus of this thesis, is one aspect of formal methods in which formal, mathematical, or analytical techniques are used in the whole design process. Formal verification is a method to verify that an implementation *satisfies* its specification. Current practice uses exhaustive or partial testing and simulation to check the correctness of a design. This practice is not acceptable for asynchronous systems due to their complexity. Formal verification complements partial testing by using more rigorous methods. Two verification methods are explored in this thesis: *state space exploration* and *proof-based* verification.

1.1 Motivation

As circuits and systems become larger and more complex, designers are forced to leave low-level design methodologies and move towards a higher level of abstraction. The final implementation can be constructed from a high level description using synthesis tools or automatic code generation, or may even be hand-crafted by experienced designers. By spending more time at the higher level, one can concentrate on the design itself, rather than the detailed implementation. This trend is also seen in software design, where programmers use CASE[1] tools instead of coding the design directly in C language.

Nowadays, hardware design resembles traditional computer programming by using hardware description languages to describe components and circuits [171]. Thus, techniques or methodologies used to verify software are adaptable to hardware designs.

Formal methods can be applied to all levels of the design process. However, formal verification is most effective if it is applied at a higher level of abstraction – i.e. communicating processes instead of gate-level components. At a higher level of

[1]CASE stands for Case Aided Software Engineering.

abstraction the complexity of the system under study is less than the complexity of the actual implementation, and is well within the capability of current tools. Application of formal methods at this higher level adds another benefit in that flaws can be discovered earlier and this results in tremendous saving of time and effort. In this spirit, it is better to model hardware design as communicating processes, rather than interconnected transistors.

A system is constructed from a set of pre-designed (and possibly pre-verified) modules or components[2]. A design is a composition of these modules. The behavior of the composition must satisfy its specification. The purpose of formal verification is to show or prove that the implementation or composition meets its specification.

In an asynchronous system, modules communicate with each other using a hand-shaking mechanism – a protocol. The behavior of the system is reflected in the protocol it implements. *Designing and verifying asynchronous circuits, for instance, become a protocol design process.* By reasoning about the protocol it implements, one can verify the behavior of an asynchronous system.

1.2 Objectives

The objectives of this thesis are:

- to identify a suitable design, validation and verification methodology for asynchronous systems (especially for asynchronous circuits)
- to present a preliminary proof that formal verification is an integral part of the identified methodology by providing verifications of several asynchronous circuits and an asynchronous systems

[2]In the sequel, the term "modules" and "components" are used interchangeably. They refer to the basic building blocks.

- to verify asynchronous systems through a high-level protocol verification.

1.3 Main Areas of Contribution

The following is a partial list of the contributions of this thesis.

1. A comprehensive overview of design methodologies for asynchronous hardware design.

2. Identification of an integrated asynchronous system design methodology using protocol engineering with PROMELA and SDL.

3. Verification of asynchronous systems through *state space exploration* with protocol verification tools. Two verification approaches are illustrated: modeling asynchronous circuits and an asynchronous system in PROMELA and SDL, and using SPIN and the SDT tool respectively to verify the circuits and system.

4. Programming techniques to model *speed-independence*, *fundamental mode*, and *input-output mode* asynchronous circuits in PROMELA.

5. Providing a novel design of a *Counterflow Pipeline Processor*, an asynchronous system, in SDL.

6. A preliminary attempt to reason about asynchronous circuits through a proof-based approach by using higher-order logic.

1.4 Structure of Thesis

This thesis is divided into two parts: introduction and discussion. In the first part, two introductory chapters are presented.

Chapter 2 gives an overview of design methodologies for asynchronous circuits.

Chapter 3 provides an overview of formal methods in hardware design.

Chapter 4 discusses the use of PROMELA to model asynchronous circuits and the use of SPIN, an automated validation tool, to verify the circuit. The approach uses a state-space exploration technique. Two different operating modes were exercised. A hazardous gate-level C-element, operating under *input-output* mode, is the circuit under study in the first case. In the second case, SPIN exercised a *fundamental* mode circuit and showed the existence of a hazard in the circuit. The hazard was removed, and SPIN was used to show the absence of hazards.

Chapter 5 discusses the use of *Specification and Description Language* (SDL), to specify an asynchronous system: a *Counterflow Pipeline Processor*. In the earlier discussions, the systems under study were asynchronous circuits. In this chapter, we show that the state-space exploration technique also applies to asynchronous systems in general.

Chapter 6 discusses initial attempts to use proof-based methodology to reason about asynchronous systems. In this chapter, Real Time CSP (RT-CSP) and higher-order logic are used to model asynchronous circuits.

Chapter 7 concludes this thesis with a conclusion, list of contributions, and list of future research.

Chapter 2

Overview of Design Methodologies for Asynchronous Circuits

The motivation of this study is to facilitate the application of formal verification methods to digital asynchronous circuits. The result of this study, however, is not restricted to asynchronous circuits. It is applicable to asynchronous systems in general. To give a foundation and perspective where our work fits in, in this chapter we present an overview of the work in the field of asynchronous circuits. This chapter is a revised version of our technical report [138].

2.1 Problems with Synchronous Circuits

Problems inherent in synchronous circuits have forced circuit designers to look for other alternatives, such as asynchronous circuits. These problems are associated with the use of a global clock, such as clock skew, worst-case behavior, scalability, power management, silicon area consumption, meta-stability of *synchronizers* and *arbiters*, and the inability of the synchronous circuits to cope with the increasing importance

of wires due to smaller circuit sizes. In the following subsections we look at these problems in more detail.

2.1.1 Clock skew

One of the most fundamental problems in conventionally-clocked synchronous circuits is clock skew, i.e. the clock signal does not arrive at the same time in many parts of a circuit. Clock skew becomes a significant problem in a large design, where a common global clock must be distributed to all parts of the circuit. Three factors that contribute to clock skew are discussed in [90]. They are:

1. RC (combination of resistance and capacitance) value of global distribution lines

2. Unequal clock paths to various modules

3. Variances in *gate threshold voltage* (V_t).

The first two factors are functions of layout and all three factors depend on the fabrication process (the technology used to implement the design). A designer may be able to control the first two factors but cannot control the third factor (gate threshold voltage).

There are various ways to reduce the clock skew problem. One approach is to use a *H-tree scheme* for distributing the clock. However, this adds an extra complication, since the designer must take into consideration the physical wire dimension (size and length), physical layout (such as transistor orientation), and the number of modules that the clock has to drive. This would be impractical for a large design.

2.1.2 Worst-case behavior

In synchronous circuits, sufficient time must be allowed between clock pulses for the slowest portion of the system to respond. Faster parts are in effect slowed down to the speed of the slowest one [93]. Therefore, synchronous circuits operate at their worst-case behavior.

2.1.3 Scalability problem

There are two meanings of scalability: scaling down feature size, or expanding a design by adding more modules (making it larger).

Shrinking feature size λ (a notation used in [174]) changes many physical parameters which determine various delays [152]. As the microelectronic technology grows, switching speed becomes closer to propagation speed of electrical signal in wires. Thus, delay in wires must be accounted for in a design. A synchronous circuit implemented in one technology cannot be implemented directly in another technology which uses a smaller feature size. Recalculating timing properties of the whole design may be needed. This becomes problematic as the VLSI technology progresses towards devices with smaller feature size.

In the context of extending a system by adding more modules, synchronous circuits cannot be extended easily. For example, if we have an n-stage synchronous FIFO, we cannot add one more stage without re-timing the circuit. Sufficient time must be allowed for the whole circuit to make necessary transitions before the next clock.

2.1.4 Power management

In a synchronous circuit, the clock runs continuously even when the circuit is not processing data. This unnecessary action draws power, up to half of the total power

in current high-speed microprocessors. These clock transitions produce heat that must be dissipated. As an example, the clock circuit in the *DEC Alpha* chip (a synchronous design) consumes more than half of its 30 watts of power [38]. Such a design may not be suitable for portable, battery-operated computers or devices.

In CMOS circuits, there is a high dynamic current during '0-to-1' or '1-to-0' transitions. This could be a problem in synchronous circuits where most components of the circuit draw power almost at the same time. This could create a spike (positive-going pulse) or a dip (negative-going pulse). Moreover, a higher-rated power supply and wider wires are needed to implement the design.

2.1.5 Area penalty

It is difficult to route the global clock in a large synchronous chip. Global clocking circuitry with its long and wide paths consume a fair share of silicon area. Again, it is interesting to note that in the *DEC Alpha* chip, 25% to 30% of the area is used for clock circuitry.

2.1.6 Wires become more important

As the size of devices becomes smaller, wire properties become more important. Sutherland and Mead in [161] observed that the cost (area and energy) and performance limitations of microcircuits are dominated by the wires rather than the switching elements. Modern integrated circuit technology has reached the point where the switching of the individual transistor is no longer significantly greater than the transition delay of a long wire. Any design methodology must incorporate the importance of wire in the model. This was not the case with earlier design methodologies commonly used in synchronous designs. In *delay-insensitive* asynchronous circuits,

wire delays do not change the correctness of the circuits.

2.1.7 Metastability of synchronizers or arbiters

A possible approach to implement a large synchronous circuit is to partition the circuit into several synchronous parts or modules. Each module is a synchronous system with an independent clock. Synchronization is done using a synchronizing element (*synchronizer*) or *arbiter*. Unfortunately this method is not reliable due to the metastability condition of the synchronizing element [153]. There is a probability of synchronization failure, which will leave the synchronizer in the metastable condition. The time required to get out of this metastable condition is unbounded. If this condition stays for too long and a time-limit is enforced, an illegal or incorrect state can result.

With all of the problems described above, the complexity of designing synchronous circuits has become unmanageable. The low-level design process is not transparent to the designer who must understand the detail and physical aspects of the circuit under design. All of these problems lead system designers to look for alternatives, such as asynchronous systems.

2.2 Advantages of Asynchronous Circuits

A natural approach to avoid the problems described in the previous section would be to use asynchronous circuits. There is no global clock to worry about. For instance, in one type of asynchronous circuit, called *delay insensitive* circuits, correct operations

are still maintained even under varying delays.

Asynchronous design methodologies are not new. In the 1950s, Muller and his colleagues at the University of Illinois developed *speed-independent circuits*, i.e. circuits that do not depend on the relative speed of their components or gates [111, 121]. Unfortunately, at that time various problems and difficulties made asynchronous circuits unpopular compared to their synchronous counterparts. A historical perspective of research in the field of asynchronous circuits is available in [172].

Some advantages of asynchronous circuits are discussed below. (A more detailed discussion is available in [62, 138].)

2.2.1 Absence of global clock

The absence of a global clock avoids problems associated with a global clock such as clock skew, as discussed earlier. The lack of a global clock also removes the routing area which was needed to distribute the clock signal to all parts of a chip. Theoretically, it should be easier to design a large asynchronous system than to design a synchronous version.

2.2.2 Average-case performance

Since faster components are not clocked with a slower clock, those components can operate as fast as they permit. Thus, asynchronous circuits operate at average-case performance.

2.2.3 Scalable

Asynchronous circuits are scalable in the sense that extending or adding new modules to an existing system does not require re-timing the whole system. As long as the

new module conforms to the protocol requirement, the system can be extended easily.

Shrinking feature size (λ) changes timing properties. However in *delay insensitive* asynchronous circuits, this does not affect the correctness of the circuit. Changes in timing properties will only change the performance of the system.

2.2.4 Better power management

In asynchronous circuits, transitions occur only when needed (i.e. when the circuit is processing data). Unused circuitry is powered off automatically. This behavior reduces power consumption dramatically and makes asynchronous circuits suitable for low-power devices, such as hand-held computers. There is an interest in low-power asynchronous circuits as witnessed by the asynchronous *ARM microprocessor*[1] developed by the Amulet group at the University of Manchester [57, 133, 134] and the low-power Error-corrector circuitry for the *Digital Compact Cassette* (DCC) developed at Phillips [165].

2.2.5 Flexible

Asynchronous circuits are flexible. Replacing slow modules or components in an asynchronous circuit, one does not have to redesign or recalculate timing properties of the whole circuit. For example, if a module is replaced by a faster version, the system will adapt and take advantage of the new speed, increasing its performance.

[1]Currently, the Apple Newton PDA uses the (synchronous) ARM processor. By using an asynchronous version of the ARM processor, they hope they can reduce power (battery) usage.

2.3 Difficulties with asynchronous circuits

From the descriptions provided in the previous section, it seems clear that the use of asynchronous circuits is the obvious solution to those problems. However, asynchronous circuits have been slow in gaining popularity. The following is a list of some possible reasons:

- **Lack of commercial asynchronous elements**. Simple asynchronous elements, such as a *C-element*, are not available off-the-shelf. This makes it difficult for practitioners to experiment with asynchronous circuits.

- **Difficult to design**. Correct asynchronous circuits are difficult to design (e.g. they must be hazard-free). A signal in a circuit may take different paths depending on the relative delay of its components. These delays may not be known or may vary with time and the design process. A designer must consider a large number of possible cases to ensure that the design operates correctly. Existing simulators only handle a small set of test cases (i.e., a set of assumed delays). It is difficult to ensure the correctness of an asynchronous circuit design in all possible cases [114]. Synchronous designs circumvent this problem by allowing changes within a sufficient time frame (between clock pulses).

- **Lack of a simple design methodology**. Ad hoc design methodologies do not work in designing asynchronous circuits, while available design methodologies are difficult and cumbersome. It was reported in [172] that an asynchronous synthesis method was actually demonstrated as early as in 1962 by Zemanek. Unfortunately it was too complex and went unnoticed.

- **Required area for handshaking circuitry**. Each module in an asynchronous circuit must indicate when it has completed its task and communicate this to

its neighboring modules. Unfortunately this handshaking mechanism requires additional area. For example, in [110] it was indicated that their asynchronous circuit requires a 40% active area penalty. However, since there is no longer a global clock, area that was required for routing the clock signal can be used for this handshaking circuitry. In any case, implementing a design with asynchronous circuits does not necessarily reduce silicon area.

- **Lack of knowledge in asynchronous design**. As previously stated, asynchronous circuits are not new. Traces of the development of asynchronous circuits can be found as early as the 1950s with Muller and his *speed-independent* circuits. However, it is only recently that designers and theoreticians have looked at asynchronous circuits with more hope. In education, synchronous circuits were the main focus of study. Only recently has the portion of curriculum dealing with asynchronous circuits been emphasized.

- **No social demand**. Synchronous circuits were easier and more accepted by the hardware designers. Industries are still reluctant to fabricate asynchronous components since there is no market for them, and it is difficult to find off-the-shelf components. As a result, companies are still reluctant to use asynchronous circuits. This cycle must be broken.

- **Asynchronous circuits have larger reachable states and are more difficult to reason about**. Asynchronous circuits are similar to non-deterministic and concurrent systems. An asynchronous design usually has a larger number of reachable states compared to its synchronous counterpart [114]. As a result, it is more difficult to formally reason about asynchronous circuits.

- **Asynchronous circuits violate design for testability and design for verifiability**. To avoid hazards, redundant implicants are usually added in circuits. However, this could make the circuit untestable. In addition, to make the circuit verifiable, complexity must be reduced. This reduction is usually done by restricting or reducing the number of reachable states through the introduction of clocked latches, a synchronous design. Such "*Design for Verifiability*," by reducing the state-space approach as was suggested in [114], makes use of a synchronous design rather than an asynchronous one.

2.4 Asynchronous Signaling Conventions

In this section we will discuss two signaling conventions commonly used in asynchronous circuits, namely *four-phase signaling* and *two-phase signaling.* Since there is no global clock, synchronization or handshaking in asynchronous circuits must follow a standard signaling convention. Some designs may use a combination of the two signaling schemes by providing signaling conversions.

If the objective is to have a faster circuit, *two-phase signaling* is usually used. If the compactness (smaller size) is the main objective, *four-phase signaling* is usually used.

2.4.1 Four-phase signaling

In *four-phase signaling*, also known as *four-cycle*, *Muller*, or *return-to-zero* [153] signaling, there are four phases ($r\uparrow, a\uparrow, r\downarrow, a\downarrow$), where r and a indicate *request* and *acknowledge* signals respectively. (Sometimes the literature uses different names, e.g. *request* and *done* instead of *request* and *acknowledge.*) A four-phase signaling timing diagram is shown in Figure 2.1 (Concentrate on the *request* and *acknowledge* signals).

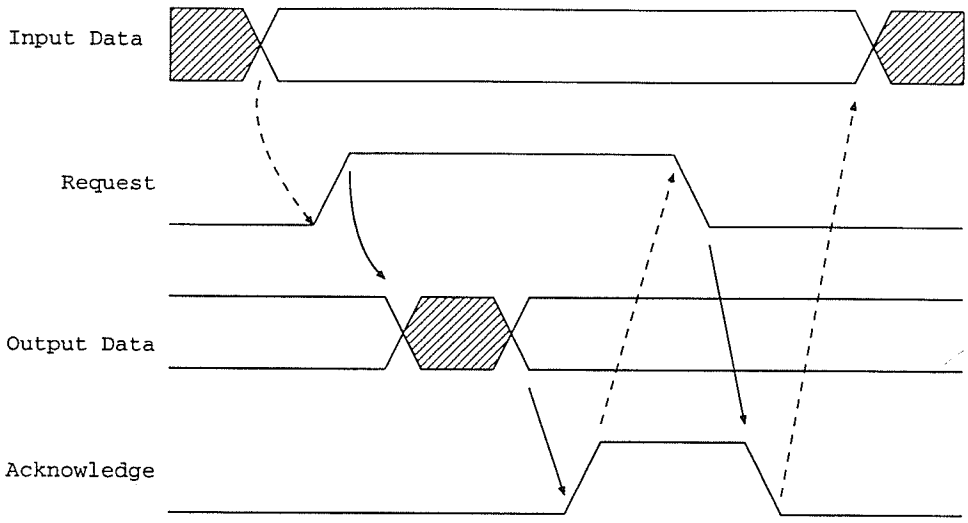


Figure 2.1: Four-phase signaling convention

The four-phase signaling operates as follows. After the input data is stable, the sender asserts a *request* signal (phase 1). After the data is accepted, this condition is then acknowledged by the *acknowledge* signal by the receiver (phase 2). The sender's *request* signal can then return to its original (zero) level (phase 3), which is again followed by the *return to zero* of the receiver's *acknowledge* signal (phase 4). Notice that there are four phases in one cycle.

One criticism of this approach is that the third and fourth step (the return to zero) are needed only to return the signals to their original levels and do not contribute to computation. Thus, time and energy to perform the transitions are wasted. However, it turns out that the implementation of four-phase signaling requires less area compared to two-phase signaling (which will be described shortly). Examples of the use of four-phase signaling are found in [47, 100, 101, 145, 166, 168].

2.4.2 Two-phase signaling

The term *two-phase* is derived from the fact that there are two phases in this handshaking mechanism; the sender's active phase (phase 1) and the receiver's active phase

(phase 2). This signaling is also known by different names, such as NRZ (*non-return-to-zero*), or *transition signaling*. A two-phase signaling timing diagram is shown in Figure 2.2. Notice that only two transitions ($r\uparrow; a\uparrow$ or $r\downarrow; a\downarrow$) are needed to complete one cycle.

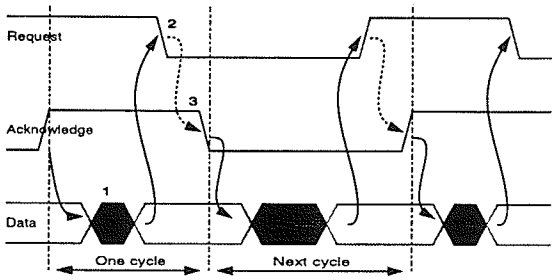


Figure 2.2: Two-phase signaling convention

In two-phase signaling, after the data is valid, the sender asserts an event (either rising or falling transition) on the *request* line and holds the data during this receiver's active phase. The receiver then sends an acknowledge event on the *acknowledge* line after the data is received. The sender is back in its active phase.

Two-phase signaling is potentially faster than four-phase signaling, at the expense of more components (thus larger area) in the implementation. Examples of two-phase signaling can be found in [62, 160].

2.5 Asynchronous Signaling with Data

In the previous section we separated data from control circuitry. We have discussed signaling conventions only for the control part of asynchronous circuits. In this section we discuss signaling conventions and their relation with the data part.

2.5.1 Bundled data

The term *bundled data* suggests that the *data wires* must be treated as a bundle. In other words, the data wires are bundled with the *request* signal, and their validity is indicated by a handshake signal whose delay is calculated based on the data wire characteristics. Thus some assumption or knowledge of the bounded delays are required. The delays in data transmission must be less than the delays in transmitting the *request* event. Usually, the delay of the request signal is determined by the worst-case delay of the data wires.

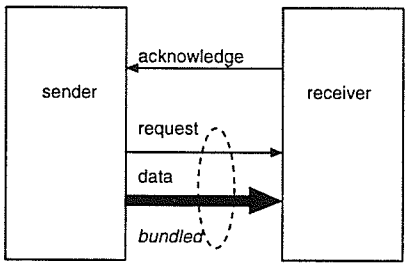


Figure 2.3: Bundled data

Bundled data is usually used in a circuit where the actual physical layout of data wires is close to the handshaking signal and they have similar characteristics. In many cases the handshaking signal must be delayed, such as by making it part of data line (the $(n+1)^{th}$ bit), and by making it longer, with a sufficient margin to make it slower than the delay in data wires. It should be noted that n data bits only require $(n+2)$ wires in such an implementation.

2.5.2 Dual-rail code

Many asynchronous circuits are implemented using a *dual-rail* or *double-rail code*, in which one bit of data is represented with two wires. Distinct lines are allocated to logical "0" and "1". (See Table 2.1.)

Table 2.1: Dual-rail code

Code	Interpretation
00	idle
01	valid "0"
10	valid "1"
11	illegal

A slightly different signaling similar to a dual-rail code is a *ternary* signaling in which "0", "1", and "*null*" $(-)$ or "undefined" (U) are used to differentiate logical "0" and "1". Data is represented by transitions from *null* to valid data and back again. Thus every transition from "0" to "1" or vice versa must go through *null*.

One criticism of a dual-rail approach is that it requires more wires (thus more area) than the other schemes. In the implementation, n data bits require $2n$ wires. An alternative solution is to encode the data electrically as three different discrete logic levels as discussed in [77].

2.5.3 Four-state code

A *four-state code* [60, 105] is similar to a dual-rail code in that two wires, which correspond to four states, are needed to represent one bit. Two states are needed to represent logical "1" (call them $P1$ and $Q1$) and the other two states are needed for logical "0" ($P0$ and $Q0$). Timing information is embedded as two alternating phases P and Q. Each data bit in a stream must have the opposite phase of the data bit preceding it. For example, the sequence "0010" is represented as "$P0, Q0, P1, Q0$". Notice that only one bit is changed between transitions. This approach is similar to another signaling called *Level-Encoded 2-phase Dual-Rail* (LEDR) [42].

Table 2.2: Four-state code

Code	Interpretation
00	logical 0 ($P0$)
01	logical 1 ($Q1$)
10	logical 0 ($Q0$)
11	logical 1 ($P1$)

2.6 Classes of Asynchronous Circuits

Based on the delay assumption or delay model, asynchronous circuits can be divided into three classes, as summarized in Table 2.3. These terms are sometimes used interchangeably.

Table 2.3: Classes of asynchronous circuits based on delay assumptions

Class	Gate Delays	Line Delays
Huffman circuit	bounded	bounded
Speed-independent	unbounded	zero
Delay-insensitive	unbounded	unbounded

A **speed-independent** circuit is a circuit whose operation does not depend on the relative speed of its components or gates. The delay of wires is assumed to be zero (i.e. lumped to the gates).

A **delay-insensitive** circuit is a circuit whose correct operation does not depend on the relative delays of its components and wires. Its behavior is retained when delays are added to its input and output wires. The class of delay-insensitive circuits is too small making it impractical for a real design. In many cases some compromises are made. For example, if an *isochronic fork*, (i.e. a forked wire where the difference in delays between destinations is negligible) is added to the *delay-insensitive* class, we have a *quasi-delay-insensitive* circuit. Theoretically, more circuits can be imple-

mented as quasi-delay-insensitive than pure delay-insensitive. However in some cases, for example in an environment where gate delays are comparable to wire delays, it might be difficult to implement isochronic forks. In general, the use of isochronic forks is restricted to small circuits or modules in a closed or local area.

The term *self-timed* is also commonly used. Self-timed refers to different things (sometimes incorrectly) in different contexts in the literature:

1. It refers to a circuit whose behavior does not depend on any relative delays among physical elements [110].

2. It relates to the completion or handshaking signals. In addition to performing computation, a self-timed circuit supplies a completion signal to indicate when the computation is done [76].

Self-timed assumes that a circuit can be decomposed into *equipotential* regions inside in which wire delays are negligible. In short, a self-timed circuit is a circuit that keeps time to itself. Sequence and time only relate inside the circuit itself [153].

Based on the interaction of the circuits with their environments, asynchronous circuits have two operation modes: *fundamental mode* and *input-output mode*. In *fundamental mode*, a circuit must be stabilized completely before another input change can be applied. In *input-output mode*, the input may change as soon as appropriate output is observed at the output port.

2.7 Asynchronous Design Methodologies

There are several asynchronous design methodologies, summarized in [138]. The choice of the design methodology depends on the underlying model (i.e., the class of asynchronous circuits used in the implementation), and the choice also depends on

the goal of the design (e.g. if speed is more important than area and power). The purest asynchronous design is usually *delay-insensitive*, and uses *dual-rail encoding*, and *four-phase handshaking*.

Some design methodologies are more expressive than others, but others may be more practical and easier to use. The choice is up to the designers and dependent upon the application. In the next subsections, we describe some of these design methodologies.

2.7.1 Macromodular

In the *macromodular* approach [32, 33, 128, 159] a designer works with building blocks called *macromodular cells* or *macromodules*. The cells are in the form of modular boxes that can be combined together in a frame structure. These cells have different functions, such as *registers*, *adders*, *memories*, *control devices*, *call*, *merge*, and *rendezvous*. The idea is that by using these building blocks, an electronically naive user could construct a special purpose and complex computer system. One could concentrate on the algorithm by manipulating the physical elements much as one would symbolically manipulate primitives in an assembly program. The designer does not have to worry about timing properties. Several "computers" have been built and dismantled to test algorithms ranging from random number generators, an FFT, to a speech pattern analyzer.

2.7.2 Micropipeline

Sutherland [160] introduces the term *micropipeline* for a simple form of event-driven elastic pipeline. He suggests the use of a *transition signaling* framework, where rising and falling transitions have the same meaning: they are *events*. This framework of-

fers the ability to build a complex system by hierarchical composition (a bottom-up approach) from simple modules which communicate with *two-phase bundled data* signaling. (For the definition of two-phase bundled data, see Section 2.5.1.) Sutherland suggests the use of a set of event-driven elements (partly shown in Figure 2.4) to compose a design.

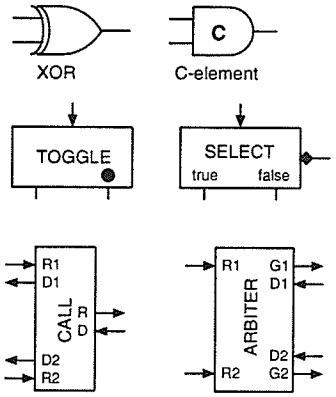


Figure 2.4: Event-driven elements

The micropipeline approach is appealing because of its practicality in by making some compromises (due to the two-phase bundled signaling). Micropipelines have been widely used as in [17, 62]. Micropipelines can also be implemented in existing FPGA technologies [54]. An asynchronous version of the ARM microprocessor is currently under development [133, 134] using the micropipeline approach.

2.7.3 Finite State Machines (FSM)

Finite State Machines (FSMs) are understood and commonly used by engineers. Therefore, it is desirable to extend FSMs into Asynchronous FSMs (AFSMs). Traditionally, asynchronous designs were specified with *Asynchronous Finite State Machines* (AFSMs).

AFSMs are usually implemented as *Huffman* circuits in *fundamental mode*, where

new inputs are applied only after the circuit has assimilated the previous input change, that is, when the memory elements are stable.

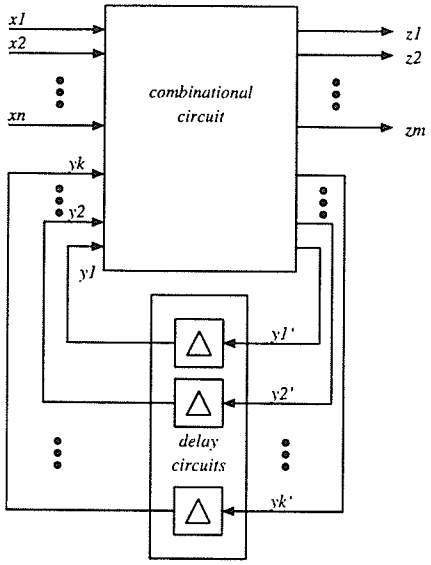


Figure 2.5: Huffman circuit

To avoid hazards, earlier AFSMs were usually restricted to *Single Input Change* (SIC) mode, in which only one (single) input change is allowed at a time and then the circuit must wait until the system is stable again before further input changes may occur. Unfortunately, this is impractical for most applications since the environment usually permits *Multiple Input Change* (MIC). A possible approach is to use *burst-mode* FSM (a limited form of MIC) in which, at a given state, an input burst is permitted. The system then generates the corresponding output burst and moves to a new state.

There are some restrictions to burst mode [125]. Firstly, no input burst in a given state can be a subset of another input burst. Secondly, a state is always entered with the same set of inputs, i.e. each state has a *unique entry point.*

Several limitations associated with the FSM approach include:

- There is an exponential dependence on the number of entries or the number of

input signals for the *flow table*[2], since it requires a listing of all input combinations in the flow table. For example, a system with three inputs requires 2^3 entries. Hence, it may be impractical for large circuits.

- An FSM can model *conflicts*, situations in which the outcome depends on the input conditions. For example, if there are more than one "next states" that can be reached from a given state in an FSM, the choice of the next state depends on the input conditions. However, an FSM cannot describe concurrent operations since, by definition, at any given time an FSM must be in only one state. Concurrency can be described by multiple FSMs. However, it is difficult to decompose an FSM into smaller concurrent FSMs.

- Special care must be taken for multiple input changes which can produce intermediate changes and hazards. A general treatment is to add redundant prime implicants to the combinational circuit.

Examples of the use of AFSMs in asynchronous circuits are illustrated in [36, 35, 125, 126]

2.7.4 Petri nets and their derivatives

Petri nets are graphical and mathematical tools for systems that are characterized as being concurrent, asynchronous, distributed, parallel, non-deterministic, and/or stochastic [122]. Thus, they provide a viable approach to specify and analyze asynchronous circuits.

Analysis of Petri nets may be done by *reachability analysis*, i.e. given an initial state, a set of possible reachable states are generated. This approach may only be

[2]A flow table is a table needed in the synthesis of an asynchronous sequential circuit.

suitable for small circuits since it enumerates all possibilities. Another problem is the difficulty of decomposing a Petri net description into smaller nets.

For an introduction to Petri nets, we refer the reader to [122, 132]. Examples of the use of Petri nets as specifications for asynchronous circuits are available in [39, 117].

Signal Transition Graph

A variant of Petri nets, called *Signal Transition Graphs* (STGs), are commonly used in asynchronous circuits. This technique was introduced by Chu in [30].

Unlike several classical approaches where a circuit is described in a state diagram, in an STG the behavior of the circuit is specified with signal transitions. A signal can make two transitions: rising and falling. In the classical approach, a circuit is represented by a vector $(x_1, x_2, \ldots, x_n)$ where $x_1, x_2, \ldots, x_n$ are signals in the circuit. This representation requires 2^n states, which grows exponentially with the number of signals (n). The complexity of the same circuit with an STG is reduced to $2n$, since only transitions are needed.

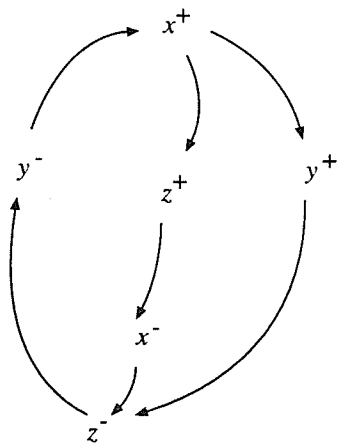


Figure 2.6: Example of a Signal Transition Graph

In an STG, a rising transition of signal x is denoted by x^+ and a falling transition

as x^-. Figure 2.6 gives an example in which the rising of x, that is x^+, causes signal y and z to go high (y^+ and z^+). After signal z has gone high, x will go low (x^-), and so forth. One can view the STG as a different form of timing diagram commonly used to design digital circuits.

STGs can be viewed as *interpreted free choice*[3] Petri nets where transitions represent changes of values of circuit signals. An STG is expressive enough for specifying concurrency and conflicts, with its timing diagram-like specification, but simple enough for analysis.

From STGs, a form of finite automata called a *state graph* can be derived. State assignments are then carried out using a simple rule. If the state assignment is consistent and no two distinct states receive the same binary representation, a speed-independent implementation can be realized. (See [30] for details.)

It should be noted that interest is mainly focused on a subset of STGs, a *live persistent* STG, which corresponds to *live persistent* state graph. In this STG, persistent means that the firing of one transition will not disable other enabled transitions. Circuits resulting from this class are *deadlock-free* speed-independent circuits.

STGs are widely used, as illustrated in [7, 29, 30, 31, 76, 82, 110, 119].

Change Diagram

Independently, in Russia, a similar approach to an STG called a *Change Diagram* was developed. The approach and tools, called *FORCAGE*, are described in [84].

[3] *Free choice* means that *conflict*, where a *token* can fire more than one transition, is allowed as long as the *place* where the conflict occurs is the only input for these transitions.

2.7.5 Process Algebras and Programming Languages

There are specification formalisms that resemble programming languages based on process algebras, e.g. *Calculus of Communicating Systems* (CCS) [115] and *Communicating Sequential Processes* (CSP) [71, 72]. These formalisms can also be used to specify asynchronous circuits.

CCS

CCS is a formalism developed by Milner [115]. Examples of the CCS-based approach for specification and verification of asynchronous circuits are presented in [10, 94, 158].

CSP

CSP is a formalism developed by C.A.R. Hoare [72]. Examples of CSP-based formalisms for designing asynchronous circuits are the work of A. Martin [100, 101, 102, 103], CP-0 [166], Tangram [167], Synchronized Transitions [92], Occam [15], and hopCP [62]. Some highlights of their methods are described in the following sections.

Martin's Production Rule

In A. Martin's work [100, 101, 102, 103], a circuit is described by a program, which is a set of concurrent processes communicating by input and output commands on channels. This description can be compiled into a delay-insensitive circuit. The compilation process involves *process decomposition*, *handshaking expansion*, *production rule expansion*, and *operator reduction*. Potentially, at each stage, a series of *semantic preserving* transformations could be applied to the description. Tools have been built to automate parts of the design procedure. The resulting output is a layout of a correct delay-insensitive asynchronous circuit with *dual-rail* and *four-phase*

handshaking.

Originally they built such circuits with a set of standard cells, such as `AND`, `OR`, `C-element`, `WIRE`, `FORK`, `ARBITER`, and `SYNCHRONIZER`. In later versions of their tools, they use "complex gates", where a gate cell is generated on the fly based on the logic function needed.

A working asynchronous microprocessor was built with this approach and it worked at the first attempt even under varying conditions [102]. A GaAs implementation of the microprocessor was also made based on the same description [164]. This proves the flexibility of asynchronous circuits, that is, they will adapt to the environment and technology used to implement them.

Tangram

Tangram is also a CSP-based language, used at Phillips for the specification of asynchronous circuits. A Tangram program is usually compiled into *handshake circuits* as an intermediate format, before the design can be realized as a full-custom delay-insensitive circuit or an implementation using FPGA technology [167].

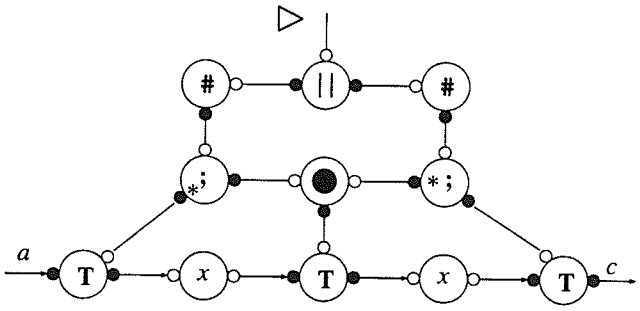


Figure 2.7: Example of a handshake circuit

A handshake circuit is a network of *handshake components*, connected by point-to-point handshake channels (see Figure 2.7). Each channel has an *active* side (represented with a small solid circle, •) at one end and a *passive* side (represented by a

small hollow circle, ∘) at the other end.

Each handshake component has a certain function. For example, a circle with a number sign (#) inside it represents a "*repeater*", which has a Tangram notation of "$a^{\circ} : \#[b^{\bullet}]$". Once it is enabled by a signal in channel a, it will generate a repeated signal in channel b. It should be noted that the Tangram approach does not commit to a specific protocol or data representation. That is, the implementation could use four-phase signaling or two-phase signaling. The uses of Tangram are presented in [166, 167, 168, 165].

PROMELA

PROMELA is also a CSP-based language developed to model computer protocols. In [141, 143, 140] PROMELA is used to model asynchronous circuits. The following is a partial list why PROMELA is suitable to model asynchronous circuits:

1. PROMELA is based on CSP, and thus has a solid foundation for formal verification.

2. The availability of validation tool SPIN provides the designers with a means with which to validate and verify their designs. SPIN provides state-space exploration verification which can be automated.

3. There is a possible translation from hardware description (i.e. netlist) to a PROMELA model, as illustrated in chapter 4.

A more elaborate discussion is presented in chapter 4.

Synchronize Transitions

Synchronize Transitions (ST) is a language that can be used to describe asynchronous

circuits/systems [155]. An example of ST notation is shown below. Let a, b, x, y, and z be Boolean state variables.

$$\ll a \neq b \rightarrow z := x \text{ AND } y \gg$$

The above description defines a state change. When $a \neq b$, z should be updated according to the AND function. In general, a transition has the following form.

$$\ll \textit{precondition} \rightarrow \textit{action} \gg$$

When the precondition is the constant TRUE, it can be omitted.

An ST design consists of a set of transitions. All transitions may execute their actions whenever they are enabled. However, an enabled transition does not have to execute immediately. The computation can be modeled as the repeated non-deterministic selection and execution of an enabled transition. This is similar to UNITY [28], a notation for describing concurrent computation based on guarded commands.

Verification of an ST design can be done by translating the description into another description that can be fed into LP, the *Larch Prover* [56]. A set of tools to perform the translation is available [155]. Another verification approach is to translate the ST description into an FL program[4] [92]. The verification is done by the FL program.

SDL

Specification and Description Language (SDL) is an industrial-strength formal language. It is defined by the ITU-T as recommendation Z.100. SDL systems can be translated into executable code. Recently, there have been a number of attempts in translating SDL systems into hardware designs through VHDL. In this thesis, we use SDL as the specification language for specifying a *Counterflow Pipeline Processor*. SDL is suitable for specifying asynchronous circuits and systems. This approach is

[4]FL is a functional language similar to the meta language ML

elaborated in chapter 5.

2.7.6 Trace theory

There are several different trace theories that have been applied to study concurrent systems. In the field of (speed-independent) asynchronous circuits, we follow Dill's version [45] as quoted from [61]:

> *"Trace theory is a formalism for modeling, specifying and verifying speed-independent circuits. Its main idea is that the behavior of an asynchronous circuit can be described by a set of* traces *(sequence of transitions)."*

A system and its elements can be specified with finite-length sequences of *atomic statements*. Such a sequence is called a *trace* and the atomic statements are called *symbols*. A specification consists of a trace set and an *alphabet* (a set of symbols). For example [61], a non-inverting BUFFER with input a and output b has a success set $\{\epsilon, a, ab, aba, \ldots\}$. An improper use of the BUFFER, which creates a "choke", is the trace $\{aa\}$.

Trace theory is appealing because regular sets are understood by many designers, and there are potentially nice compositional properties [45]. Dill has written a verifier to verify speed-independent circuits in a post-design stage of a design.

A different approach is taken by Ebergen [47] in which trace theory is used to guarantee the condition of decompositions in his correct-by-construction design methodology; with his automated tool called VERDECT [49].

The general idea behind the use of trace theory is that one can decompose a specification into specifications of smaller circuits with the constraint that it must preserve the original specification irrespective of delays in connection wires. Once

the decomposition process is done, the circuit can be realized by "basic" (standard) circuits. The resulting circuits are delay-insensitive.

There are a number of variations of trace theory. For example, *directed trace structures*, that is, trace structures with inputs and outputs, were studied in [169, 170]. In these structures, whenever an output is connected to an input, an implicit delay is placed between the two.

The use of trace theory in VLSI was introduced in [169] and applied to asynchronous circuits in [45, 47, 61, 144, 168, 170].

2.8 Applications of Asynchronous Circuits

Asynchronous circuits have a promising future due to their potential features. To date, there have been a number of real implementations including:

- The Caltech asynchronous microprocessor [102, 164], and the asynchronous ARM microprocessor by a group in Manchester [57, 130, 133].
- Implementations in Field Programmable Gate Arrays (FPGAs) using Actel [18, 62], Montage [69], and Xilinx [54, 97].
- A Communication chip [99], and a Post Office chip [158].
- Neural network implementations [124].
- Wavefront arbiters [61].
- Error-corrector circuitry for Digital Compact Cassettes (DCC) [165].

Most of the implementations are still for experimental purposes. However, they have shown their potential for industrial applications.

2.9 Summary of Chapter 2

An overview of design methodologies for asynchronous circuits is presented in this chapter. Some of the surveyed design methodologies are practical but they do not support formal methods. Other surveyed methodologies support formal methods but they are either too complicated for practical purposes, or they are not ready for public usage. The availability of tools is another important aspect when selecting a methodology. It is as important as the availability of compiler tools in the widespread take-up of high-level languages [70].

The use of PROMELA and SDL, which can be used to describe asynchronous circuits at a high-level of abstraction, is the selected approach in this thesis. This approach, by using protocol engineering, is elaborated in chapter 4 and chapter 5.

The main contribution of this chapter is its extensive overview and references to design methodologies for asynchronous circuits. This overview includes major asynchronous design methodologies, design tools, and examples of actual implementations.

Chapter 3

Overview of Formal Methods for Hardware Design

> *Non-exhaustive testing can be used to show the presence of bugs, but never to show their absence.* (E. W. Dijkstra)

The development of hardware and software technology has resulted in complex hardware and software designs. Traditionally, simulations have been the only technique to check the correctness of a design. Unfortunately, for complex designs, this *ad hoc* technique is becoming impractical. Exhaustive testing for hardware and software is expensive and impractical, if not impossible [22, 44]. Still, software and circuits must be tested, especially if they are going to be used in critical applications, such as in a heart pacemaker, a digital *fly-by-wire* control system [44], or in a missile guidance system. The ability to find errors as early as possible is also desirable, as this can reduce cost tremendously in commercial designs.

Recently, there has been a trend towards formal approaches in hardware and software design. Formal methods use mathematics or mathematical analysis techniques in the development of a design. Exhaustive and partial simulations are replaced or augmented with mathematical proofs or a systematic state space search. This chap-

ter provides an overview of several formal approaches in hardware design. A survey of formal methods in hardware design is available in [26, 66, 106, 157] and a related topic is discussed in [51, 149].

In this thesis, the study is oriented towards hardware design. Fortunately, formal methods in hardware design have had more success as compared with that in software design [156]. This is due to the fact that:

1. The size of the problems in hardware verification are usually smaller than to that of software. Hardware problems are well within the capabilities of existing theorem proving tools.

2. Hardware usually implements relatively simple algorithms, leaving complex algorithms to their software counterparts.

3. Hardware designers are familiar with modular designs with a small restricted and well tested (verified) set of libraries as their building blocks. Individual components in the library can be verified separately, reducing the complexity of the whole design.

One serious obstacle to the use of formal methods is that mathematical expertise is required, especially if a proof-based method is chosen. This is a problem since many practitioners have limited formal education in computer science and mathematics. The acceptance and success of formal methods in industry depends on the methodology itself. The methodology should not be difficult to learn and it must be supported by industrial-strength tools.

3.1 Design Process

Designing a system involves three main components: *specification*, *implementation*, and *verification & validation*. Their interactions are shown in Figure 3.1.

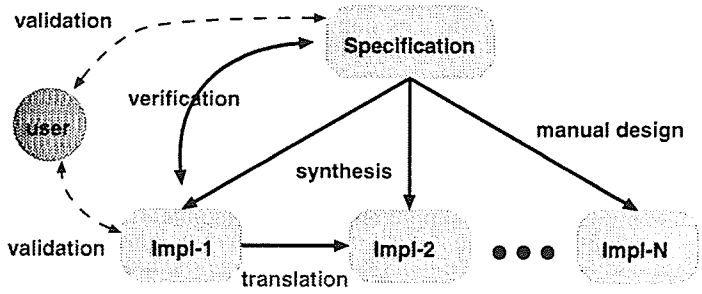


Figure 3.1: Design Process

A designer starts the process by describing what the system is intended to do. This description is the *specification* of the system. An *implementation* is then produced based on the specification. Specification and implementation have different levels of abstraction. The implementation can be automatically generated (called *synthesis*) or hand-crafted by an experienced designer.

One specification can lead to several possible implementations, called *alternative implementations*. It is also possible to *translate* an implementation to another implementation at the same abstraction level. An example of this action occurs when an optimization (e.g. area or speed optimization) is performed.

The next step is to check whether or not the implementation meets the specification and intentions. This can be done by simulation, formal verification, and validation. The main goal of verification is to show that the implementation indeed satisfies its specification. Validation, on the other hand, refers to human interaction where the designer checks whether or not the implementation satisfies the user's or designer's intention. It is possible that an implementation meets its specification,

but the specification itself is at fault or does not capture what the designer wants. This is possible due to unreadable or obscure notation used in the specification or miscommunication between the specifier and the client.

An implementation can become a specification for a subsequent lower-level implementation. This process is repeated until the actual design (e.g. a layout in hardware design or an executable program in software design) is produced. We elaborate on some on these components in the following sections.

The use of formal methods in hardware (VLSI) design can be divided into two classes. In the first class, a provable correct design is generated from a formal specification. In the second class, a post-design verification is performed. The goal of the latter is to guarantee the correctness of existing designs. These existing designs are either still in use or will be reused in a subsequent system.

3.2 Formal Specification

In the context of hardware design, a *specification* is a description of what a hardware circuit must do without constraining how it is achieved (implemented). Specification can be done informally with explanations in a natural language (such as in plain English sentences) or formally with formal languages (which usually have a machine-checkable syntax), Hardware Description Languages (HDL), or mathematical notations.

Informal specification can lead to ambiguous interpretations of imprecise or poorly-worded descriptions. In order to avoid this problem, specification must be done formally, as it serves as the basis or reference for implementation, verification & validation, as well as documentation.

3.2.1 Choosing a Specification Language

Choosing a suitable specification language presents a problem. In general, there is no notation that can satisfy all possible applications. Different applications require different appropriate notations [14]. Aspects associated with selecting a specification language that have to be considered include:

- efficiency,
- expressiveness,
- abstraction and hierarchical capabilities,
- ease of automation or mechanical proof.

To date, there are numerous specification languages or notations for hardware, such as VHDL [75], Verilog [163], CIRCAL [112, 113, 114], ELLA [120], to name a few. Some of these languages are supported by modern Computer Aided Design (CAD) tools.

One must realize that some of these HDLs are not considered formal systems, i.e. one cannot reason about the design in that HDL directly. A transformation of the description into a formal system is required. For example, in [171] the argument is made that "*VHDL has no formal mathematical semantics as part of its definition, hence, programs written in it have not been amenable to formal analysis.*" He studied a subset of the VHDL language which was then translated into HOL [65], a higher-order theorem proving environment. Formal verification was then carried out within HOL.

There is also an argument that the specification should be done in a formal system (such as HOL). Once the system is verified, it is translated to an HDL for synthesis. Unfortunately, translation from a formal specification to a less formal HDL involves

some risks. Translation by using formal specifications down to the lowest level, (e.g. transistor level) should be avoided. In [79], Joyce *et al.* suggest leaving the low-level implementation (synthesis) to modern and mature CAD tools.

Another notation that has been used to specify hardware design is **Z**, in which errors in some floating-point implementations have been revealed [3]. **Z** has also been used to verify the microcode of the Inmos T800 FPU [78].

In many cases mathematical notation is easier to read (although Lamport [91] claimed that the structure is the most important factor). Consider the following example, Fermat's Last Theorem (due to [91]):

> There do not exist four positive integers, the last being greater than two, such that the sum of the first two, each raised to the power of the fourth, equals the third raised to that same power.

It is easier to read (although English is still partially used) as:

> There do not exist integers such that $x^n + y^n = z^n$, where $x, y, z > 1$ and $n > 2$.

Formal specification is not limited to computer science, engineering, and mathematics, but also has applications in other fields. For example, in chemistry, one may find the following notation easier to understand than a description in plain English (two parts of Hydrogen combined with one part of Oxygen yields two parts of water):

$$2H_2 + O_2 \rightarrow 2H_2O \tag{3.1}$$

3.2.2 Formalism for asynchronous circuits

Asynchronous circuits correspond to concurrent, communicating, and distributed systems since components in asynchronous circuits run concurrently creating nondeter-

ministic interleaved actions and interactions. Consequently, formalisms that support concurrent, communicating, and distributed systems are also suitable for asynchronous circuits. The following is a partial list of formalisms used in concurrent systems.

- **Actor**. This is a formalism suitable for asynchronous systems, although its application to asynchronous circuits has not been investigated. Actor is described in [1].

- **CCS**. CCS has been used to express asynchronous circuits [10, 94, 158], however communication in CCS is inherently synchronous [1].

- **Higher-order logic**. In [89], FAUST (a first-order theorem prover built on top of the HOL system) is used to show the existence of a hazard (i.e. to check the safety property). However, the authors specifically added delay elements (i.e. the intention is not to verify a delay-insensitive circuit).

- **Temporal Logic**. Bochmann [11] is probably the first to use temporal logic for describing the behavior of asynchronous circuits. The verification of the circuit is done through reachability analysis. The temporal ordering of actions is an important property in asynchronous circuits. Some properties (e.g. safety) of asynchronous circuits can be specified in temporal logic [11] and verified with a model checker as is done in [16, 46, 116].

- **CSP**. CSP-based techniques are quite popular in the field of concurrent systems. This approach has been adapted to asynchronous circuits, as in the works of Martin [100, 101, 103], Ebergen [48], van Berkel's Tangram [166, 167, 168], and *hopCP* [62].

- **Protocol-based formalism**. Components or modules in asynchronous circuits communicate through a handshaking mechanism: a protocol. Therefore, formalisms that can be used to specify computer protocols can also be used to specify asynchronous circuits. Examples of this approach are Murφ, AT&T COSPAN, PROMELA, and SDL. The use of PROMELA is elaborated in Chapter 4, while the use of SDL is elaborated in Chapter 5.

3.3 Synthesis and Implementation

Synthesis refers to the *automatic* generation of a circuit from a specification. The resulting circuit is then claimed to be *correct by construction.* However, since bug-free software cannot be guaranteed, there is a chance that errors are produced during synthesis. To avoid this, synthesis tools must go through the same rigorous verification. Misunderstanding and misuse of synthesis tools may also contribute to errors [59]. Therefore, post verification is still an important activity, even for automatically synthesized circuits.

Hand-crafted implementations are susceptible to design errors, especially in large designs. Thus post verification is obviously needed for this type of implementation.

3.4 Validation and Verification

Validation usually refers to human interactions where one declares an implementation as valid based on some subjective tests or simulations. Why not test and simulate only? First of all, testing and simulation are not scalable. An increase in complexity makes exhaustive testing impractical. For example, a circuit with a 16-bit input requires 2^{16} test patterns to perform exhaustive testing. For sequential circuits, the

number of test patterns increases since the order of test patterns is also crucial. Since exhaustive testing is impossible, usually designers resort to a set of test cases which are selected based on heuristic or probabilistic methods. For a critical system, this may not be acceptable since there is a possibility that a subtle error may not be discovered by the test cases. For example, the recently discovered Intel Pentium FPU bug theoretically manifests itself once every 9 to 10 billion operand pairs [175], and yet the bug hit a user not long after the chip was introduced.

Verification, on the other hand, is a *formal* (i.e., analytical) demonstration that an object (implementation) correctly meets its specification. Verification may even be used before any implementation exits and verification considers all possible situations. Verification attempts to overcome the deficiencies in testing and simulation. One way to put it is [106]:

> *"The difference between formal verification and simulation is similar to the difference between deriving laws in physics from first principles and performing experiments."*

Unfortunately, performing a formal verification on complex or VLSI-size circuits can be difficult. A design strategy, *Design for Verifiability*, is then advocated. This strategy integrates verification into the design process. In essence, the designers are restricted to well known *constructions* which are known to simplify the potential behavior of the hardware being designed [114], e.g. the use of a global clock to reduce the state space. Unfortunately, this approach is too restrictive and cannot be used for designing asynchronous circuits.

Validation and verification are not academic exercises anymore. They have increasing industrial importance, as shown by the bug found on the Pentium chip [98].

3.5 What can be verified?

Verification can be applied to the specification and to the implementation. In the former case, we want to make sure that the specification is indeed what we want. This can be done by "executing" the specification (for specifications that can be executed), or by providing consequences or proofs of certain properties. In the latter, verification is done to make sure that the implementation "satisfies" the specification. The term "satisfies" can be interpreted in several ways (depending on the application):

- The implementation exhibits the exact same behavior as the specification (i.e. they have the same state space.) This can be checked with *bisimulation* or by proving tautology. This strict an equivalence may be too restrictive for some applications.

- The implementation is more general than the specification. That is, the implementation has larger input and output sets (spaces) as compared with the specification. The implementation is capable of handling more input combinations than the specification.

- The implementation implements only a subset of the specification (i.e. the implementation has a smaller input set). However, for inputs that are not in the input set, it does not exhibit erroneous outputs. It does not produce anything. This type of implementation may be suitable for some applications.

Properties that can be verified:

- Functionality. The functionality verification ensures that the implementation exhibits the expected functionality.

- *Safety* and *liveness* properties. The *safety* property indicates conditions that should not happen. For example, for an `ARBITER`, the safety property guarantees

that only one task may be granted the *access* signal at a time. The *liveness* property guarantees that something good will happen. For example, if an *access* signal is granted, then eventually the task will access and then release the resource.

- Timing properties. Timing properties are needed for performance guarantees, or time-critical applications, or for fault tolerance.

3.6 Formal Verification Methods

Formal verification methods follow three main paths: *state-exploration* methods, *logic* or *proof-based* methods by using automated theorem provers, and *symbolic simulation* [66]. These paths are elaborated in the following sections.

3.6.1 State-space exploration method

In this approach, the implementation is translated into a state-transition structure, e.g. a finite state machine (FSM) or an automata. State exploration algorithms (e.g. *reachability analysis* or *model checking*) are used to show that the behavior of the implementation satisfies the specification. If the specification is violated, usually a trace of a counter-example can be shown.

The model in this approach is restricted to be finite-state. The biggest drawback of this method is the possibility of having a *state-space explosion.* This is a serious problem in a system where multiple components operate in parallel. As a result, the number of states grows exponentially with the number of components. Although various reduction techniques can be applied to reduce the complexity, methods that rely on explicit state representation suffer from the state explosion problems. We will

discuss such an approach in Chapter 4, where PROMELA, a language used to model computer protocols, is used to *specify* asynchronous circuits. Reachability analysis using SPIN (an automated tool) will be used to verify the circuits.

Another example of reachability analysis is the verification of *synchronized transition* (ST) programs, which can be used to describe asynchronous circuits [155]. This is achieved by translating an ST description into a functional language FL program. The reachability analysis is carried out by the FL program. Yet another example of finite state machine verification is the PROLOG program called *Verify* [6, 5].

A different type of state-exploration method is *temporal logic model checking* [34, 95]. *Temporal logic* can be used for specifying propositions which change over time without introducing time explicitly. Verification is done through *model checking* in which the implementation is described as labeled transitions (such structures are called *Kripke models*), and the possible transitions are explored [24]. An algorithm (e.g. CTL model checking or a Binary Decision Diagram-based technique) is then used to verify the system [95]. In the field of hardware design, a technique to extract the Kripke structure from a gate-level description of a circuit is explained in [46].

An advantage of the state-exploration methods is that they can be automated and need little human intervention. Although state-exploration methods could be used to verify a large state space system (e.g. a system with 10^{20} states or more [21]), fundamentally it is restricted to smaller sizes due to the state-space explosion problem.

3.6.2 Logic or proof-based methods

In this approach, the specification and implementation are translated or transformed into some form of formal logic (see Figure 3.2). At this point, the verification is carried out by using a *checker* or an automated theorem prover, for example by

checking for logical equivalence of the two logic representations [139]. Ideally, given a set of axioms, the theorem prover could come up with a proof or counter-proof that the specification and implementation logics are equivalent or that the implementation implies the specification.

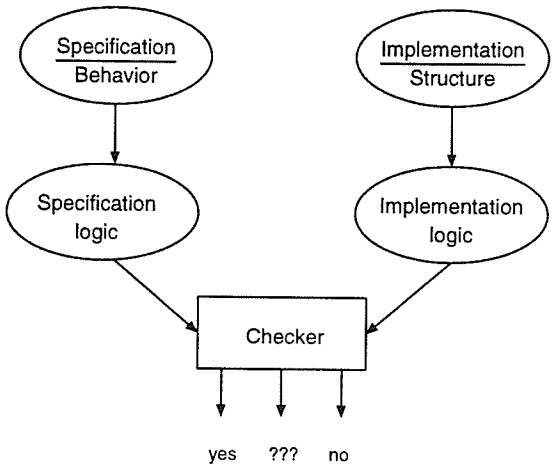


Figure 3.2: Proof-based verification process

This approach is based on *mathematical logic*, in which a formal language is used to represent logical truths and logical methods. The application of mathematical logic to digital circuits is a technological "accident", where the two symbols ("0" and "1") happen to be used to represent signals in the circuits. The "0" symbol can be used as a synonym for FALSE and "1" for TRUE. Consequently, mathematical logic can be used to study gate-level digital circuits [66].

In state-exploration methods (described previously) the main interest is the behavior or functionality of the system, whereas the proof-based approach seems to work better with the structure of the circuit [55]. An advantage of proof-based methods is that the properties of systems with infinite states can be proven by using *induction principles*. There is a strong claim that hardware verification needs more than model checking [148].

One major disadvantage of a proof-based method is that it requires a great deal of user interaction, sophistication and an understanding of both theorem proving and the structure of the circuit under investigation. It is unfortunate that many practitioners (circuit designers) do not have this background.

There are several available logics used for hardware verification. For a more detailed description, refer to standard texts on mathematical logic, such as [9]. The following is a list of examples of logics for hardware specification and verification:

1. **First-Order Logic** (FOL). FAUST, a first-order theorem prover built on top of HOL, has been used to verify digital circuits [87, 88, 89]. OTTER is another FOL theorem prover [25].

2. **Boyer-Moore Logic**. Hunt, in [74], verified the FM8501, a 16-bit microprocessor similar in complexity to a PDP-11 with Boyer-Moore logic. German and Wang used Boyer-Moore logic to formally verify parameterized hardware designs [58].

3. **Higher-Order Logic**. There have been a number of specifications and verifications with higher-order logic. Hanna and Daeche [67] were the first to use higher-order logic to verify hardware designs with their VERITAS system. HOL [65], a theorem proving environment which supports higher-order logic, seems to be a popular tool. Examples of higher-order logic usage in hardware design are provided in [23, 63, 80, 87, 88, 89, 114, 147]. Complex circuits such as microprocessors have been verified with HOL (e.g. in [80]), as has the VIPER microprocessor in [37, 96]). There are other theorem proving systems that support higher-order logic, such as VERITAS [68], PVS [41, 40], EHDM [146], and Isabelle [129].

3.6.3 Symbolic simulation

In *symbolic simulation*, a simulator is used to perform verification. In addition to "0", "1", and "X" (undefined), the simulator also accepts Boolean variables [66]. There is a variation of this new emerging approach, called *symbolic trajectory evaluation* integrated in a formal system called Voss [151].

The main advantage of symbolic simulation is the avoidance of explicit state-transition graphs. Thus it will reduce the complexity and should be able to handle large systems.

3.7 Summary of Chapter 3

Formal verification is one aspect of formal methods. Three verification methods are available: state-space exploration, proof-based , and symbolic simulation. In this thesis, the main focus is on formal verification through state-space exploration. This approach is selected because it can be automated. The selected design methodology, verification through protocol engineering with PROMELA and SDL, supports the state-space exploration method. This approach is elaborated in chapter 4 and chapter 5. An initial attempt at proof-based verification is also presented in chapter 6.

The main contribution of Chapter 3 is an overview of formal methods specifically applied to hardware design.

Chapter 4

State Space Exploration-based Verification of Asynchronous Circuits

Verification of asynchronous systems is difficult due to concurrency which can result in a large state space. In this chapter we explore the verification of asynchronous circuits based on state exploration techniques. The circuit under study is modeled in a language called PROMELA and an automated tool called SPIN is used to verify the circuit. Two types of asynchronous circuits are modeled and verified in this chapter: a *fundamental mode* circuit and an *input-output* mode circuit.

4.1 Introduction to PROMELA and SPIN

PROMELA is a language developed by G. Holzmann [73] to model computer protocols. It is based on Hoare's CSP [72]. In this chapter we use PROMELA to model asynchronous circuits.

SPIN is an automated tool used to verify models written in PROMELA. It performs state space exploration or *reachability analysis*. One can write certain conditions that have to be checked by SPIN. SPIN is then executed to show the correctness of the property, or to provide a counter-example if the property is incorrect.

The main contribution of this chapter is an exploration of computer protocol development tools and the extension of their application to asynchronous circuits. These efforts were among the first attempts in evaluating and demonstrating their use in modeling and verifying asynchronous circuit implementations. A more compact representation of this chapter has been published in [141].

Compared with other surveyed methodologies, discussed in chapter 2, PROMELA is selected due to the following practical reasons:

1. PROMELA is based on Hoare's CSP and it is amenable to formal methods.

2. The language PROMELA is similar to the C language which is commonly used in the industry. Therefore, the learning curve to master PROMELA should not be too steep.

3. The availability of automated tools, such as SPIN, helps in automating the validation and the verification of circuit designs.

4. PROMELA abstracts a design into communicating processes (higher abstraction) instead of gate components (lower level of abstraction).

5. There is a straight forward translation from a gate-level netlist (a list of components and their connections) into PROMELA statements.

4.2 Modeling and Verification of an Input-Output Mode Circuits

One of the difficulties in designing asynchronous circuits is to guarantee a hazard-free circuit. A hazard is a special type of race. A *race* is a condition whereby several signal values are changing due to an input transition. Races that can lead to a state differing from the predicted one are called *critical races*. In a sequential circuit, this could lead to a steady-state error. It should be noted that critical races are well known problems in concurrent systems [43, 81]. This correspondence illustrates the relationship between asynchronous circuits and concurrent systems.

A number of hazard analysis techniques have been proposed, such as the use of three-valued simulation [50], multi-valued simulation [2], algebras [53, 104, 107], and the use of trace structures [45]. In this section and its subsections we show the use of PROMELA and SPIN to find hazards in a C-element, a commonly used component in asynchronous circuits.

4.2.1 Circuit Under Study

A C-element is an element commonly used in asynchronous circuits to merge events (also called a `MERGE`, `JOIN`[1], or `RENDEZVOUS`). The C-element has been selected as the circuit under study due to its common usage in asynchronous designs. The Truth Table of a 2-input C-element is shown in Table 4.1. In this table, z_i indicates the current output value, and z_{i+1} indicates the next output value. A possible gate-level implementation of this C-element is shown in Figure 4.1.

It is trivial to verify, with the no-delay (zero gate and wire delay) assumption, that

[1]A JOIN has a more restricted behavior compared to a C-element in that once an input is asserted, it cannot be withdrawn.

Table 4.1: Truth table of a 2-input C-element

x	y	z_{i+1}
0	0	0
0	1	z_i
1	0	z_i
1	1	1

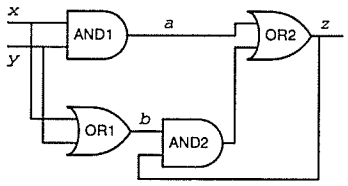


Figure 4.1: Gate level implementation of 2-input C-element

the circuit functions as a C-element. Unfortunately, if we use the *speed-independent* model – where the gate delays are unknown – and the *input-output* mode of operation, this implementation contains a critical race which leads to incorrect behavior.

The explanation is as follows. Suppose gate `OR1` is much slower compared to the other gates, or path (wire) b is much longer compared to the other wires. We start with "0" at both input x and y. Let the inputs x and y both change from "0" to "1" ($00 \rightarrow 11$) simultaneously. Output z will change from "0" to "1", due to path a (but assume that path b is still "0"). Suddenly x changes back to "0" ($11 \rightarrow 01$). It is possible that path a will become "0" before path b becomes "1" resulting in a "0" at the output z, which is an error. This scenario is summarized in Table 4.2. It has been proven in [19] that there is no hazard-free implementation of a C-element using only 2-input gates.

Table 4.2: Value changes in the C-element shown in Figure 4.1

x	y	a	b	c	z
0	0	0	0	0	0
1	1	1	0*	0	1
0	1	0	1	0	0

Note: 0* = about to change from "0" to "1"

4.2.2 PROMELA Model

A PROMELA model of our C-element is shown below. Text between the symbols /* and */ is a comment.

```
/* declaration */
bit a, b, c, x, y, z, oldz;
bit wait;

/* macros */
#define AND(x,y,out) (out != (x&&y)) -> out = x&&y
#define OR(in1,in2,out) (out != (in1||in2)) -> out = in1||in2

/* main block */
init
{
        run stimulus();
        run monitor();
        do
        :: AND(x,y,a) /* AND1 */
        :: OR(x,y,b) /* OR1 */
        :: AND(b,z,c) /* AND2 */
        :: OR(a,c,z) /* OR2 */
        od
}
```

The model looks somewhat like a C program. It begins with declarations of signals or variables used in the circuit. In this example we use global declarations. These variables form the state of the circuit. The declarations are followed by *macros* (indicated by the `define` directive) to implement logic functions (`AND`, `OR`). Macros are similar to macros in a C program, since the source text is processed by a C preprocessor. These macros will be elaborated upon in the next few paragraphs. The "`init`" keyword is comparable to the function "`main()`" in a standard C program, inside of which we have two parallel processes, indicated by the "`run`" keyword, and the connections of `AND` and `OR` gates which form the implementation.

In PROMELA two colons (`::`) indicate choices, one of which will be selected on each cycle. In the example section we have four choices: `AND1`, `OR1`, `AND2`, and `OR2`.

On each cycle one of these will be selected and executed, causing the output value of the gate to be evaluated. This technique provides non-deterministic delays to each gate and path, modeling *speed-independence.*

A statement in PROMELA is either executable or blocked depending on the values of variables in the statement. For example, in the statement "$P \rightarrow Q$", Q will be executed if P is evaluated to true. P is a statement, but it can also be viewed as a condition or a guard. In the earlier fragment, the `AND` macro is executed if "`(out != x && y)`" is evaluated to true, that is if the output of an `AND` gate must be changed. The "`&&`" indicates a logical `AND` operation. The *do-loop*, indicated by the keywords "`do`" and "`od`," represents a repetition. In our model we have four options within the *do-loop.* Only one gate is selected arbitrarily for execution at a time. The `AND` and `OR` macros are written in this way to give "equal" execution cycles to all gates that need to be evaluated. That is, if the output of a gate is correct, it does not need to be evaluated. It sets the condition or guard to be false and that statement will not be selected during the current cycle. This technique is used to avoid a condition in which one gate is selected and evaluated infinitely often while the rest are not evaluated, resulting in an artificially incorrect behavior. This is possible due to the lack of a fairness property in PROMELA.

The "`stimulus()`" process supplies input patterns to the circuit. Care must be taken in this process. Inputs must not change until the effect of previous input change is observed at the output (modeling *input-output mode*). Otherwise, if inputs change too fast and are not absorbed by any gate, this situation may produce another artificially incorrect behavior. It should be noted that the effect at the output may be due to one or a combination of several paths. Other paths may still be in transition (in an unstable state) leading to races and hazards if they exist. This approach corresponds to the *input-output* mode of the circuit. The technique could be implemented

by setting a flag, called "*wait*" (the name is unimportant), which indicates that the system has absorbed the inputs and is ready for another input change. The process "`stimulus()`" must wait until the "*wait*" flag is set to "0". Once the "`stimulus()`" receives this condition, it toggles the value of x or y and sets the *wait* flag to "1".

```
proctype stimulus()
{
        do
        :: wait == 0 ->
                oldz=z;
                if
                :: x = 1-x
                :: y = 1-y
                fi;
                wait = 1
        od
}
```

System monitoring is done by another process, called "`monitor()`" in this implementation. This process waits until the output is correct for the current inputs and then sets the "*wait*" flag to "0".

```
proctype monitor()
{
        do
        :: wait == 1 ->
                if
                :: (x==0 && y==0 && z==0)
                :: (x==1 && y==1 && z==1)
                :: (x!=y && z==oldz)
                fi;
                wait = 0
        od
}
```

Process "`monitor()`" starts when "*wait*" is set to "1". It will then wait until one of the three conditions is true, that is until an appropriate output has been observed. It will then continue by setting the "*wait*" flag to "0", letting the "`stimulus()`" process change the inputs. Notice that we have only conditions in the *if* - *fi* block

since in PROMELA there is no difference between conditions and statements. For example, we can view the statement "`(x = 1-x)`" as a condition which is always true, i.e. assignments always produce a true value.

To check the correctness of the behavior of our circuit we are going to use a "*temporal claim.*" Before we do that, we have to identify what are considered correct or incorrect behaviors for our circuit. Informally, a 2-input C-element has the following properties:

1. if both inputs x and y have the same value, propagate the value to the output z.

2. if the values of x and y are different, that is `(x != y)`, then the output z must hold its previous value `(z == oldz)`.

Any correct implementation of a C-element should not violate the above properties. A race, for example, could violate the above properties. We will show that our implementation of the C-element (shown in Figure 4.1) violates the second property.

Before writing the specification of the C-element in PROMELA, an introduction to specifying correctness requirements in PROMELA is in order. In PROMELA, we can formalize *livelocks* (something cannot happen infinitely often) with a special PROMELA label called "*acceptance-state label.*" An acceptance-state label is a label which is started with the character sequence "*accept.*" We can use *accept0*, *accept1*, or *acceptable* as acceptance labels. To quote from [73]:

> An acceptance-state label marks a state that may not be part of a sequence of states that can be repeated infinitely often.

We will use acceptance-state labels in the specification of our C-element.

A correctness requirement in PROMELA can be done through a "*temporal claim.*" Temporal claims define temporal ordering of properties of states. To illustrate the use of a temporal claim, we borrow the example from [73]. We can use a temporal claim to express the requirement that "every state in which property P is true is followed by a state in which property Q is true," by using the following statement:

```
P -> Q
```

All correctness criteria in PROMELA are based on properties that are claimed to be *impossible.* The notation for temporal claims is

```
never {...}
```

The temporal claim is a finite state machine (FSM), that is why it is also called a "*claim FSM.*"

The requirement of the previous example is written as

```
never {
   do
   :: skip
   od;
   P -> !Q }
```

The above temporal claim states that "it is impossible for a state in which P is true to be followed by a state in which property Q is false." The "*skip*" statement in the above temporal claim is needed so that the claim is independent of the initial sequence of events. If it is omitted, the requirement is matched only when, at the first state the property of P is true and, at the second state the property of Q is false. The requirement cannot be matched if, at the first state the property of P is false, but at the second state P is true and at the third state Q is false. That is, the incorrect condition is not at the beginning of the sequence. This is not what we want.

By having the "*skip*" statement, the sequences where the truth value of P changes from true to false a few times are permitted by the above claim.

An automatic validator, such as SPIN, matches the proposition in the temporal claim (*claim FSM*) in every state of the system. As the system state changes, the validator also moves the state of the claim FSM from one proposition to the next one. To quote from [73]:

> To match a temporal claim, at every state in a sequence of states, the proposition at the corresponding state in the claim FSM must be true.

If the temporal claim is fully matched, i.e. the temporal claim is completed, then a behavior which is claimed to be impossible is found.

The properties of a 2-input C-element can be rewritten as:

1. if both inputs x and y have the same value, then the value of the output z should not be different than the input.

2. If the values of x and y are different, then the value of the output z should not change.

The temporal claim of the above properties can be written as follows.

```
never {
        do
        :: skip
        :: (x != y) -> goto accept0
        :: (x == y) -> goto accept1
        od;
accept0:
        do
        :: (z != oldz)
        od;
accept1:
        do
        :: (z != x || z != y)
        od
}
```

The explanation of the above claim is as follows. The "`skip`" in the first *do-loop* matches anything, i.e. the condition is permitted to hold in any number of initial steps. If the "`skip`" is omitted, then the proposition is only matched against the first two reachable states.

At some non-deterministic time, either proposition "`(x != y)`" or "`(x == y)`" in the claim FSM can be matched. Depending on the proposition matched in the previous state, we then move to either "*accept0*" or "*accept1*." For example, if in the previous state the proposition "`(x != y)`" is matched, then the claim FSM is moved to the "*accept0*" label. Here, we use the *acceptance-state label* to indicate that the claim FSM cannot remain in this state infinitely often (*livelock*). In other words, we want to express that if the values of x and y are different `(x != y)` then the output should not change or the system cannot remain infinitely often in a state where the proposition "`(z != oldz)`" is true. (Compare this with the second property of the C-element.) If at the "*accept0*" label the statement "`(z != oldz)`" is not executable, then the proposition is false, which means the system behaves correctly `(z == oldz)`. We can truncate the search at this path and start looking at other reachable system states. If at the "*accept0*" label the statement "`(z != oldz)`" is true infinitely often (livelock), then an error (called an *acceptance cycle*) is found.

4.2.3 Verification Results

A complete PROMELA model of our C-element was verified with SPIN on a UNIX workstation. An *acceptance cycle* was found. The final state shows the following variables:

```
wait = 1
a = 0
b = 1
c = 0
```

```
oldz = 1
x = 0
y = 1
z = 0
```

From the above result we can see that (`x != y`) but `z` did not retain its value (`z != oldz`). An incorrect behavior is found. The values of the variables are the same as those shown in row 3 of Table 4.2. Curious readers can use SPIN to trace the execution and monitor the variables.

A common technique used to overcome critical races is to include additional prime (redundant) implicants. Unfortunately, it has been proven that there is no implementation of a hazard-free C-element from only 2-input gates [19].

4.2.4 Other C-elements

In this section we show various implementations of C-elements and verify their properties by modeling them in PROMELA and verifying the models with SPIN.

A possible implementation of a hazard-free C-element was proposed by Oleg Mayevsky [83], and it is shown in Figure 4.2. It has the following equations:

$$\begin{aligned} y_1 &= x + y \\ y_2 &= not(x \cdot y \cdot y_3) \\ y_3 &= not(y_1 \cdot y_4) \\ y_4 &= not(y_2 \cdot y_3) \\ z &= y_2 \cdot y_4 \end{aligned} \tag{4.1}$$

The circuit was modeled in PROMELA and verified with SPIN. (The complete PROMELA model is listed in Appendix A.) This implementation was found to be

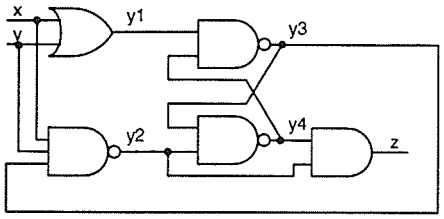


Figure 4.2: Hazard-free C-element

hazard-free. This does not violate the result in [19] as it uses a 3-input gate.

Another C-element implementation that is commonly used is shown in Figure 4.3. This circuit has the following equation [107]:

$$z' = a \cdot b + a \cdot z + b \cdot z \tag{4.2}$$

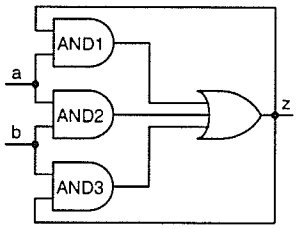


Figure 4.3: Hazardous C-element

Unfortunately, this implementation contains a critical race in the speed-independent model. It could be easily verified that signal a and b transitions from $00 \rightarrow 11 \rightarrow 01$ could result in an incorrect behavior. For example, if gate AND3 is slower compared to the rest of the gates, a transition from all "0" to all "1" (i.e., $00 \rightarrow 11$) could go through gate AND2, while the output of gate AND3 is still "0". If this condition is followed by a $11 \rightarrow 01$ transition, the circuit could produce a "0" at the output, an incorrect behavior. (The PROMELA model is listed in Appendix B.)

A summary of the above result is given in Table 4.3. A more elaborate description is found in [143].

Table 4.3: Summary of input-output mode C-element verification

Circuit	critical race
Figure 4.1	yes
Mayevski (Figure 4.2)	no
Majority (Figure 4.3)	yes

4.3 Modeling and Verification of a Fundamental Mode Circuit

In this section we show the use of PROMELA and SPIN to find a hazard in a *fundamental-mode* circuit, i.e. the circuit must be stabilized completely before another input change can be applied.

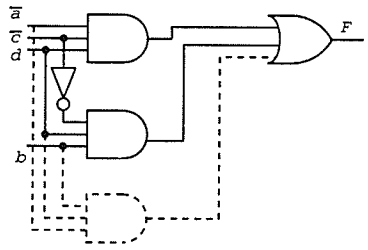


Figure 4.4: A hazardous fundamental-mode circuit

cd \ ab	00	01	11	10
00				
01	1	1		
11		1	1	
10				

Figure 4.5: K-map of circuit under study

Figure 4.4 shows a circuit that contains a hazard. Its Karnaugh map is shown in Figure 4.5. (For now, ignore the dashed lines and gate.) The circuit has the following Boolean function: $F = \bar{a}\bar{c}d + bcd$.

The following explanation shows that the implementation is not a hazard-free one. During a signal c transition, there is a possible hazard generated depending on the delays of the inverter and wires. A possible scenario is shown in Table 4.4. In this situation, $\bar{c}$ (indicated as "cbar" in the Table) is changing from "1" to "0" while the other input signals are constant. As shown in rows 5 and 6 of Table 4.4, there is a momentary dip (negative-going pulse), or hazard, at the output (F).

Table 4.4: Hazardous transitions of the circuit under study

row#	abar	cbar	b	d	n1	n2	c	F
1	1	1*	1	1	1	0	0	1
2	1	0	1	1	1*	0	0	1
3	1	0	1	1	0	0	0*	1
4	1	0	1	1	0	0	1	1*
5	1	0	1	1	0	0*	1	0
6	1	0	1	1	0	1	1	0*
7	1	0	1	1	0	1	1	1

4.3.1 PROMELA model

The PROMELA model of our circuit is as follows.

```
/* declarations */
bit a, abar, b, c, cbar, d, f;
bit oldf; /* old value of d */
bit n1, n2;
bit newp; /* new pattern */

#define AND3(x,y,z,out)  (out != (x&&y&&z)) -> out = x&&y&&z
#define OR2(x,y,out)     (out != (x||y)) -> out = x||y
#define OR3(x,y,z,out)   (out != (x||y||z)) -> out = x||y||z
#define INV(in,out)          (out != (1-in)) -> out = (1-in)

proctype netlist()
{
   do
   ::
      if
```

```
      :: AND3(abar, cbar, d, n1);
      :: INV(cbar, c);
      :: AND3(b, c, d, n2);
      :: OR2(n1,n2,f);
      fi;
      newp=0
   od
}

proctype stimulus()
{
   do
   :: timeout ->
      atomic {
      newp=1;
      oldf=f;
      if
      :: abar = 1-abar
      :: b = 1-b
      :: cbar = 1-cbar
      :: d = 1-d
      fi;
      }
   od
}

init
{
   atomic{abar=0; cbar=0; b=0; d=0; newp=1 };
   atomic {
      run stimulus();
      run netlist()
      }
}

never {
   do
   :: skip
   :: (newp==0 && oldf != f) -> break /* transition */
   od;
   do
   :: ((newp==0) && (oldf != f))
   :: ((newp==0) && (oldf == f)) -> break /* spike/dip */
   od
}
```

The description begins with declarations, followed by the actual implementation which consists of connections of the gates. This is done by using a process called "`netlist()`" (the name is not important). Double colons (`::`) indicate options and the "*do-od*" construct indicates repetition. The "*if-fi*" construct selects one of the options. It indicates that on each cycle, one of the gates is selected and is evaluated. This process is repeated until all gates are stabilized, and then the process "`netlist()`" is blocked.

The next description is a process called "`stimulus()`". (Again, the name is not important.) This process is started with the special PROMELA keyword "*timeout*," which is set to false (by the PROMELA definition of *timeout*) until there is no more action in the model at which time it will be set to true. This technique implements the *fundamental mode* operation of the circuit, i.e. it waits until the circuit stabilizes completely, then sets the "`newp`" flag to "1" (indicating a new pattern has been received), memorizes the output value "`f`" into "`oldf`" (`oldf=f`), and changes the input by toggling one of the input variables.

The "`never {...}`" construct is the specification (*claim FSM*) that we want to verify. In our particular case, we want to express the non-existence of hazards. The explanation of this *claim FSM* is as follows. At some non-deterministic time, the proposition "(`(newp == 0) && (oldf != f)`)" can be matched, i.e. there is a transition at the output `(oldf != f)` without a new input pattern being applied `(newp == 0)`. We then move to the second *do-od* loop in the *claim FSM*. We stay there until a new pattern is applied (which will set "`newp = 1`"), or there is no new input pattern but the output changes back to its previous value `(newp == 0)&&(oldf == f)`. In the first case, a new input pattern is applied `(newp = 1)`, and the search through this path is truncated because none of the statements in the second do-loop is executable. We restart the claim FSM. In this case, the circuit does not produce a hazard. In the

second case, a hazard is detected and the *temporal claim* is completed. This means that the claim FSM is matched and an incorrect behavior is detected. The above claim corresponds to a *static hazard*, i.e. for one input change there is a momentary spike ($0 \rightarrow 1 \rightarrow 0$) or dip ($1 \rightarrow 0 \rightarrow 1$) generated at the output.

4.3.2 Verification Results

The complete model was executed by SPIN on a UNIX workstation, and a hazard was found. The final state shows the following values, which are the values of row 7 in Table 4.4.

```
n1 = 0
n2 = 1
_p = 0
abar = 1
cbar = 0
oldf = 1 [previous value of F]
a = 0
b = 1
newp = 0
c = 1
d = 1
f = 1    [value of F]
```

4.3.3 Hazard Removal

A common method to combat static hazards is to add redundant prime implicants. In this particular circuit we can add another term, indicated by the dashed lines and gate in Figure 4.4. With the additional term, the Boolean function becomes $F = \bar{a}\bar{c}d + bcd + \bar{a}bd$. This additional term corresponds to the dashed line in the Karnaugh map shown in Figure 4.5.

The additional term was added to the original PROMELA model and the new model was verified through a full state space search. No hazard was found.

4.4 Summary of Chapter 4

- The use of PROMELA as a specification language for hardware designs was presented.
- Verifications of several asynchronous circuits through state space exploration were demonstrated.
- Programming techniques to model input-output mode, fundamental mode, and speed independence in PROMELA were developed and were presented.

The main contribution of this chapter is the adaptation of protocol specification and verification methodology to the specification and verification of asynchronous circuits. This work is one of the first attempts to use PROMELA and SPIN for the specification and verification of hardware designs. Possible follow-up projects would include the extensions of these techniques to larger circuits or systems, and the development of graphical interface tools to design PROMELA systems (similar to the graphical tools used in the designing of SDL systems which will be discussed in the following chapter).

Chapter 5

Design and Analysis of a Counterflow Pipeline Processor

In Chapter 4, we presented the verification of several asynchronous circuits through state-space explorations. In this chapter we extend the methodology to specify and verify an asynchronous system. The system under study is the Sproull's *Counterflow Pipeline Processor* (CFPP) [154]. We use the *Specification and Description Language* (SDL) as the specification language. Simulations are done through *Message Sequence Charts* (MSCs), which illustrate causal relationships. Validation and verification are done with the SDT[1] validation tool, which performs state-space explorations. The concept of this chapter has been published in [142].

5.1 Introduction to SDL

SDL [27] is a specification language commonly used in telecommunications. It is standardized by the ITU-T (formerly called CCITT) as the *Recommendation Z.100.*

[1]SDT is an SDL tool developed by Telelogic.

SDL has two representational forms: graphical (*SDL-GR*) and textual (*SDL-PR*). The basis for describing SDL behavior is *Extended Finite State Machines* (EFSMs), which are represented by SDL *processes*. Communications between processes and the environment are done through *signals*. For a more detailed description of SDL, we refer the reader to [8, 52, 127, 142].

Compared to the other methods surveyed, SDL is selected as the specification language in this approach for the following reasons:

1. *High-level abstraction.* The availability of *Abstract Data Types* (ADT) in SDL makes it possible for designers to concentrate on the communication aspects of the system instead of dealing with low-level bits and bytes.

2. *Graphical representation.* The availability of user friendly graphical representation in SDL (SDL-GR) is an appealing factor to many designers. The graphical representation is also generally accepted for documentation purposes.

3. *Quick learning curve.* Unlike the other formal systems surveyed, such as HOL [65] or PVS [41, 40], SDL has a relatively quick learning curve. While there may be more expressive design methodologies or specification languages, most of them tend to be too mathematical, too difficult to use, and they require a great deal of training.

4. *Availability of industrial-strength tools.* One of the most important factors of a successful design methodology is the availability of industrial-strength tools that can cope with real designs. There are at least three commercial industrial-strength SDL tools: *SDT* from *Telelogic*, *GEODE* from *Verilog*, and *MELBA* from *Collaborative Information Technology Research Institute* (Australia).

5. *Well proven.* SDL has been used to specify commercial applications by companies such as AT&T and Siemens.

The semantic models of SDL are suitable for asynchronous systems. To quote from [136][2]:

> *"In SDL, a system is described as asynchronous communicating extended finite state machines... In an SDL system description, the communication between structural elements may include delay of arbitrary length."*

SDL is perfect for specifying *speed-independent* asynchronous circuits, which are the natural implementations of the CFPP. A speed-independent circuit is a circuit that operates correctly regardless the delays of its components.

We use *SDT* version 3.0, an SDL tool developed by *Telelogic* [162], on a Sun workstation running *OpenWindows*. The SDT tool allows designers to work with the graphical (SDL-GR) and the textual (SDL-PR) representations of SDL systems. The SDT tool has various useful features; such as the ability to view the behavioral aspects of SDL systems through *Message Sequence Charts* (MSCs), and the ability to validate SDL systems through state space explorations. Version 3.0 of the SDT tool implements SDL-92 [52], which extends the SDL language with an object-oriented design methodology.

5.2 Counterflow Pipeline Processor

In [154], Sproull et al. elaborate on a novel processor architecture called the "*Counterflow Pipeline Processor*" (CFPP or SCPP). This processor has a regular structure consisting of *stage blocks* and *processors* which are connected in a pipeline fashion

[2]They identified this description in a different context, but it applies well to the present case.

as illustrated in Figure 5.1. Each *stage* block consists of an *instruction register* and a *result register*, which are shown in Figure 5.2. CFPPs have a unique feature, in that instructions and results flow in opposite directions. When an *instruction* meets a *result* in a *stage block*, both the instruction and the result are sent to the processor connected to the *stage block*. The processor executes the instruction, and then passes a modified or a new *instruction* as well as the *result* back to the *stage block*. There is a variant of CFPP called SCPP-A [118] in which some of these operations are simplified.

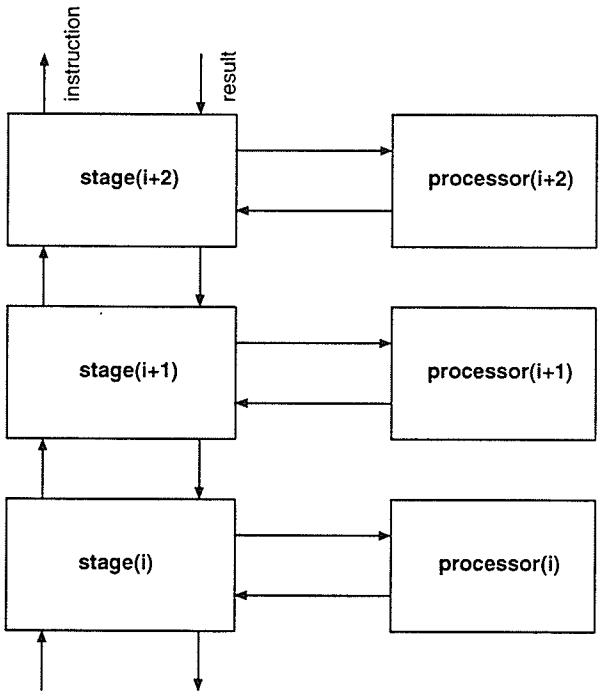


Figure 5.1: Counterflow Pipeline Processor architecture

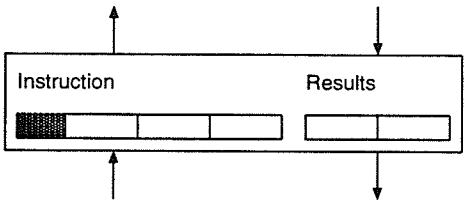


Figure 5.2: The contents of the stage block

The nature of CFPP lends itself to asynchronous implementations, i.e. local communications are used instead of a global clock. Possible asynchronous implementations of the SCPP-A are described in [86, 176]. However, their design methodologies only deal with the control part of the CFPP, whereas our approach with SDL incorporates the data part in the design. Depending on the variant of the CFPP, the flow of *instructions* and *results* may depend on the contents of the instruction and the result registers. The benefit of our approach is the ability to design and analyze a complete system. For example, one can visually inspect (with *Message Sequence Charts*) the flow of data, or one can study the effects of an execution of a certain instruction. This visual inspection ability is important in a real environment.

5.3 SDL specification of the CFPP

We make assumptions that the *instruction register* contains fields for three accumulators[3], and that the *result register* contains fields for two accumulators. This assumption is shown in Figure 5.3. We assume we have three accumulators: "A", "B", and "C".

Instruction Register

Opcode	Source1	Value1	Valid1	Source2	Value2	Valid2	Dest	Value3	Valid3

Result Register

Result1	Value1	Valid1	Result2	Value2	Valid2

Figure 5.3: Register layout

The CFPP system is partitioned into manageable functional modules: the *proces-*

[3]In this chapter, the term "accumulator" refers to a register within the instruction or the result register.

Table 5.1: The fields of the instruction register

Fields	Type	Description
Opcode	Charstring	instruction/opcode
Source1	Character	operand 1
Value1	Integer	value of operand 1
Valid1	Boolean	validity flag of operand 1
Source2	Character	operand 2
Value2	Integer	value of operand 2
Valid2	Boolean	validity flag of operand 2
Dest	Character	destination/result accumulator
Value3	Integer	value of destination accumulator
Valid3	Boolean	validity flag of destination accumulator

sor block and the *stage block*. These modules communicate with each other through a protocol which is to be developed. When needed, data (the value of the instruction and the result registers) are passed as parameters.

We use a data structure similar to the data structure suggested in [154]. The data structure can be easily implemented in SDL using the "`NEWTYPE`" construct, which resembles a *record* (in the Pascal language) or a *struct* (in the C language). The signal specifications (shown in Figure 5.4) show, among other things, the descriptions of two new types: *InstType* (for instruction type) and *ResType* (for result type). These two types are used for passing parameters in signals *inst*, *res*, and *procData*. The *InstType* type has a set of fields and types listed in Table 5.1. For example, the first field of the *InstType* type is an "*Opcode*" which has the SDL *Charstring* (character string) type. An example of data of type *InstType* is

```
(. 'LDD', 'A', 0, false, 'B', 0, false, 'C', 24, true .).
```

Similarly, the *ResType* type has two result registers: *result1* and *result2*. The explanation is similar to the previous, hence it is omitted.

```
SIGNAL
procData(InstType,ResType),
procReq, procDone,
inst(InstType), Res(ResType),
instReq, readyforInst, instReceived,
resReq, readyforRes, ResReceived;

signallist sctrl = instReq, readyforInst, instReceived,
resReq, readyforRes, ResReceived;

NEWTYPE InstType
  STRUCT
  Opcode  Charstring;
  Source1 Character;
  Value1  Integer;
  Valid1  Boolean;
  Source2 Character;
  Value2  Integer;
  Valid2  Boolean;
  Dest    Character;
  Value3  Integer;
  Valid3  Boolean;
ENDNEWTYPE InstType;

NEWTYPE ResType
  STRUCT
  Result1 Character;
  Value1  Integer;
  Valid1  Boolean;
  Result2 Character;
  Value2  Integer;
  Valid2  Boolean;
ENDNEWTYPE ResType;
```

Figure 5.4: Signal descriptions of the CFPP

The SDL system description of a CFPP with three stage blocks and three processors is shown in Figure 5.5. This figure shows three stage blocks of type *SType* and three processors of type *PType*. (This is an example where we use the SDL-92 notation.) *Signal lists* and *signal types* are shown in Figure 5.4. A *stage* block is connected to its neighboring *stage* block (upper or lower) by two channels: *data channel* (*dlow*) and *control channel* (*clow*). Data channels can carry signals which contain data, such as the *inst* signal (a signal which contains an instruction as a parameter) and the *res* signal (a signal which contains results). Control channels can carry the signals listed in signal list *sctrl*: *instReq* (instruction request), *readyforInst* (ready for instruction), *instReceived* (instruction is received), *resReq* (request to transfer result), *readyforRes* (ready for result), and *resReceived* (result has been received). In

this example, a *signal list* is used to make the diagram cleaner. By using a *signal list*, we can use the word "*(sctrl)*" instead of listing all the signals in the diagram.

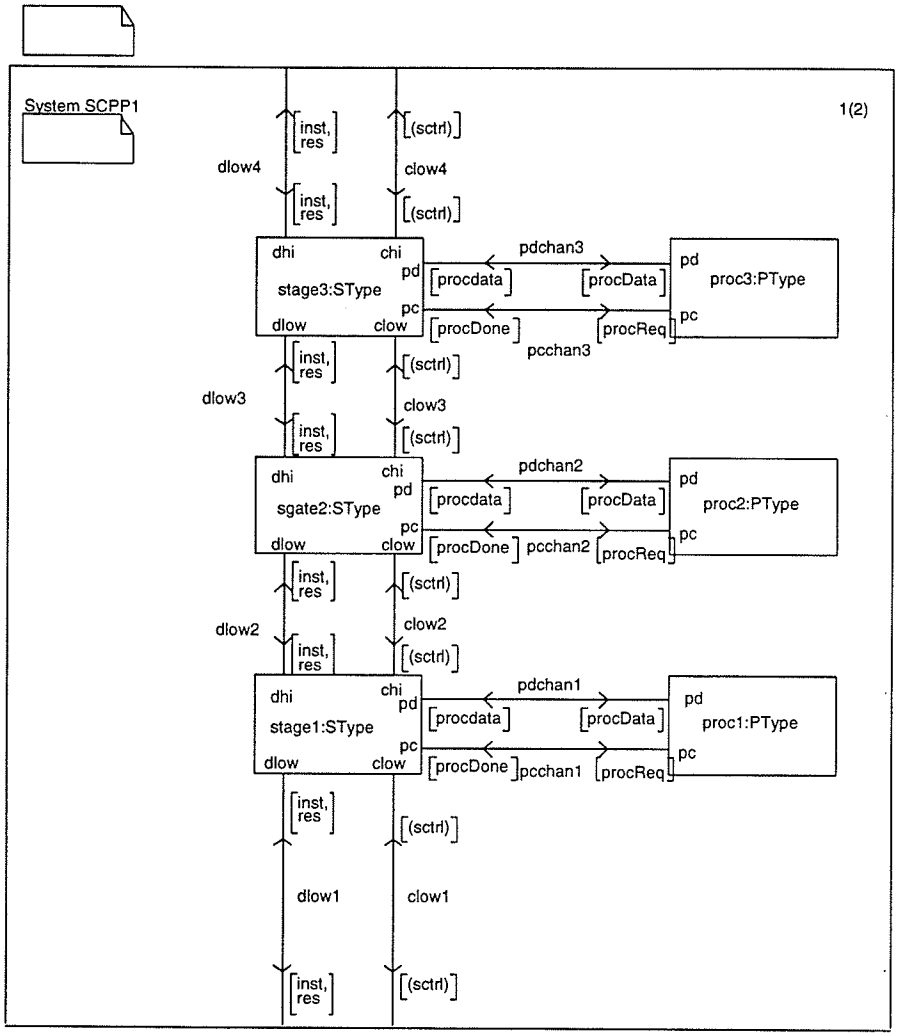


Figure 5.5: SDL system description of a three-stage CFPP

We depart from the original structure of the CFPP (as shown in Figure 5.1) in that each stage block does not have separate channels for instructions and results. We could create separate channels, but that would increase the complexity of the diagram. The separation between the data and control channels seems to be a logical approach.

5.3.1 Processor Specification

The operation of the processor is as follows. The processor accepts a request (*procReq* signal) and data (the contents of the instruction and result registers via a "*procData(IX,RX)*" signal) from its neighboring *stage* block. It then executes the instruction and possibly modifies the *result* (RX) and *instruction* (IX) values. Once the execution is completed, the results are passed back to the original *stage* block by sending a "*procDone*" signal and a "*procData(IX,RX)*" signal. *Two-phase* handshaking is used in this particular case, as the *stage* block does not send an acknowledgment that a result has been received. An example of a *procData* signal is shown below.

```
procData((. 'NOP', 'A', 0, TRUE, 'B', 1, TRUE, 'C',2,FALSE .),
(. 'A', 0, FALSE, 'B', 0, FALSE .))
```

The processor block has a *PType* block type, as shown in Figure 5.6. The interface to the outside is done through "*gates*" *pc* and *pd*. This block has one process: *P1* of type *PTGeneral*.

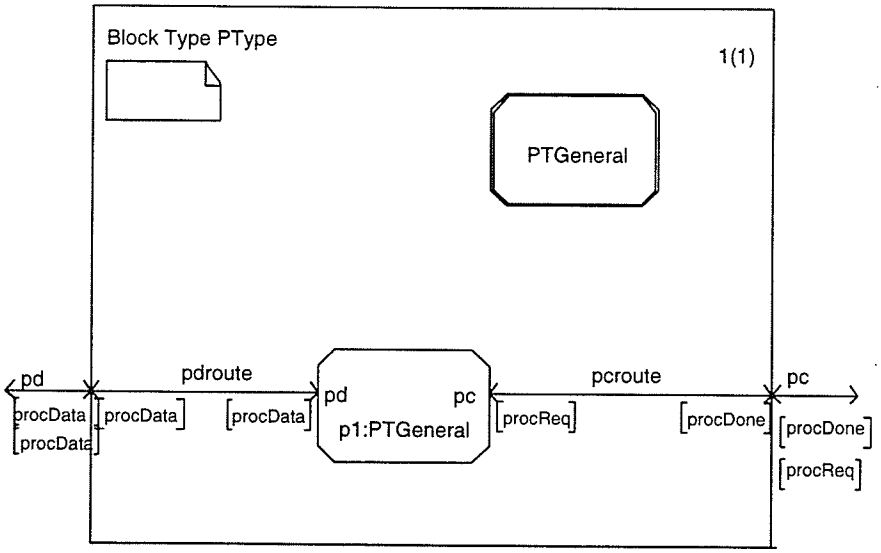


Figure 5.6: Description of block type PType

Process type "*PTGeneral*" is shown in Figure 5.7. In a process type we describe the operations associated with a *process* in an EFSM. The algorithm of process type *PTGeneral* is as follows. The process begins in the *idle* state where it waits for an incoming *procReq* signal while the other signals are saved (indicated by the asterisk). After a *procReq* signal is received, the process waits for a *procData* signal and assigns incoming parameters to local variables *IX* and *RX*. The process then calls subroutine *ProcessInst*, in which the process executes the instruction indicated by the opcode field as shown in Figure 5.8. After processing the instruction, the process sends a *procDone* signal, and also sends the resulting data to the sender with a *procData(IX,RX)* signal. The process then goes back to the *idle* state.

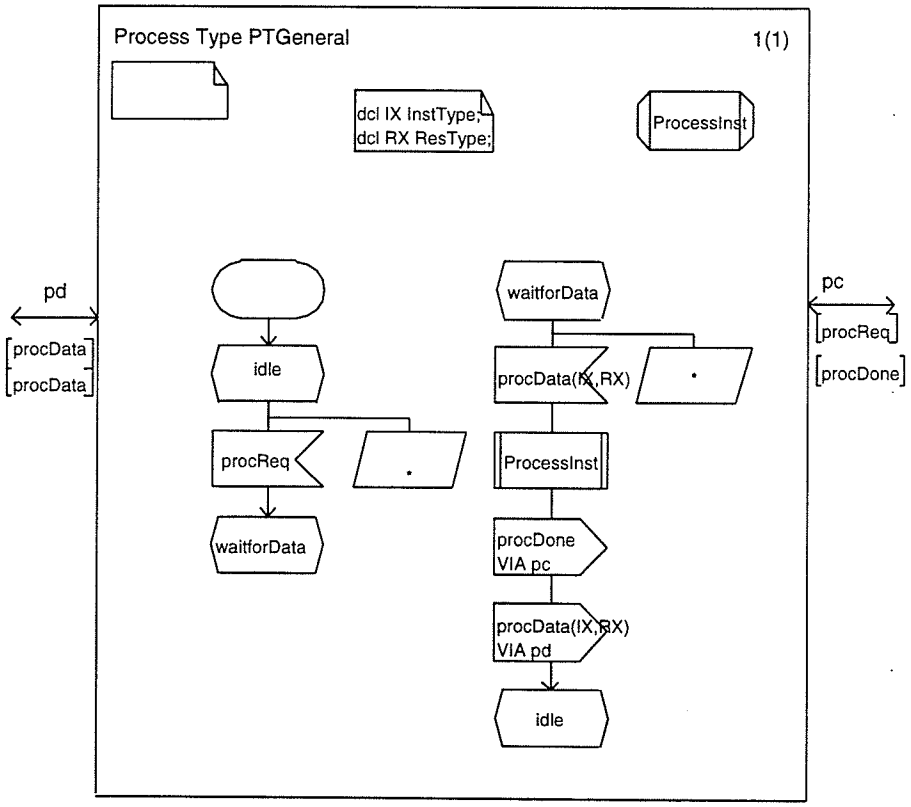


Figure 5.7: Description of process type PTGeneral

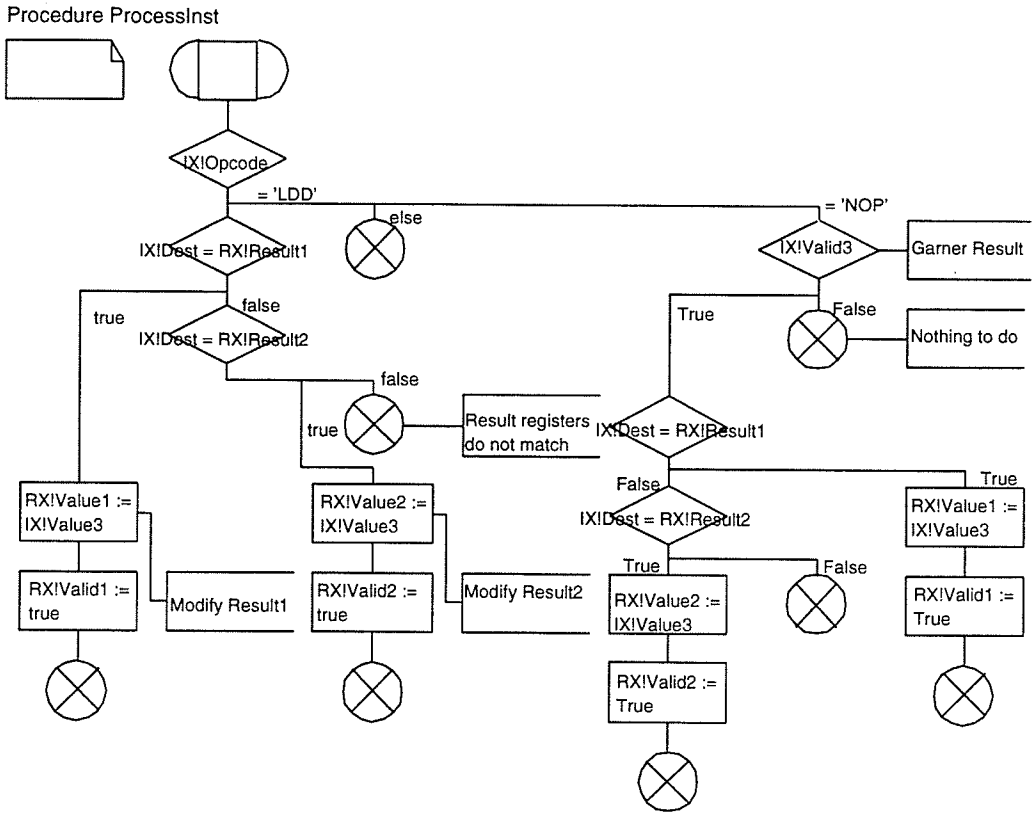


Figure 5.8: Description of procedure ProcessInst

5.3.2 Instruction Set

The instructions and actions in the *ProcessInst* subroutine are shown in Figure 5.8. The procedure *ProcessInst* defines the instruction set of the processor. In this particular example we only have two instructions: "`LDD`" and "`NOP`". An informal description of the instructions is as follows.

- `LDD`. This instruction copies the value of the accumulator stored in the "*Dest*" field to either result accumulator "*Result1*" or "*Result2*" if the name of the accumulator contained in "*Result1*" or "*Result2*" matches with the name of the accumulator contained in "*Dest*". For example, the instruction:

  ```
  (.  'LDD', 'A', 0, false, 'B', 0, false, 'C', 24, true .)
  ```

will set accumulator "*C*" to 24 if the result register contains accumulator "*C*" such as the instructions

```
(.  'C', 0, false, 'A', 0, false .)
```

or

```
(.  'A', 0, false, 'C', 0, false .)
```

- `NOP`. This instruction does nothing. However, if the *instruction* and *result* registers meet and the *result* register contains an accumulator with an invalid value, but the *instruction* register contains the same accumulator name with a valid value, then the value of the accumulator in the *result* register will be set to the valid value.

The above instructions are selected only to illustrate the operations of our CFPP. More instructions and actions (such as additions, multiplications, and other higher-level instructions) can be added by modifying this procedure. However, as the number of instructions increases, the diagram becomes more populated and complicated. In this case, the use of the textual representation of SDL (SDL-PR) would be more productive. Another option to describe more instructions would be to spread the diagram into multiple pages. The *SDT* tool provides an interface to the textual representation of SDL, although we have not explored this path.

An MSC simulation trace of the processor is presented in Figure 5.9. Two instances are shown: *env_0* (environment) and *p1_1* (process p1). We sent a *procReq* signal and a *procData* signal with its parameters from the environment. The instruction was "LDD" which should set the value of register "C" to 24 and its valid flag to true. The processor responded with a *procDone* signal and a *procData* signal. Notice that the value of register "C" became 24.

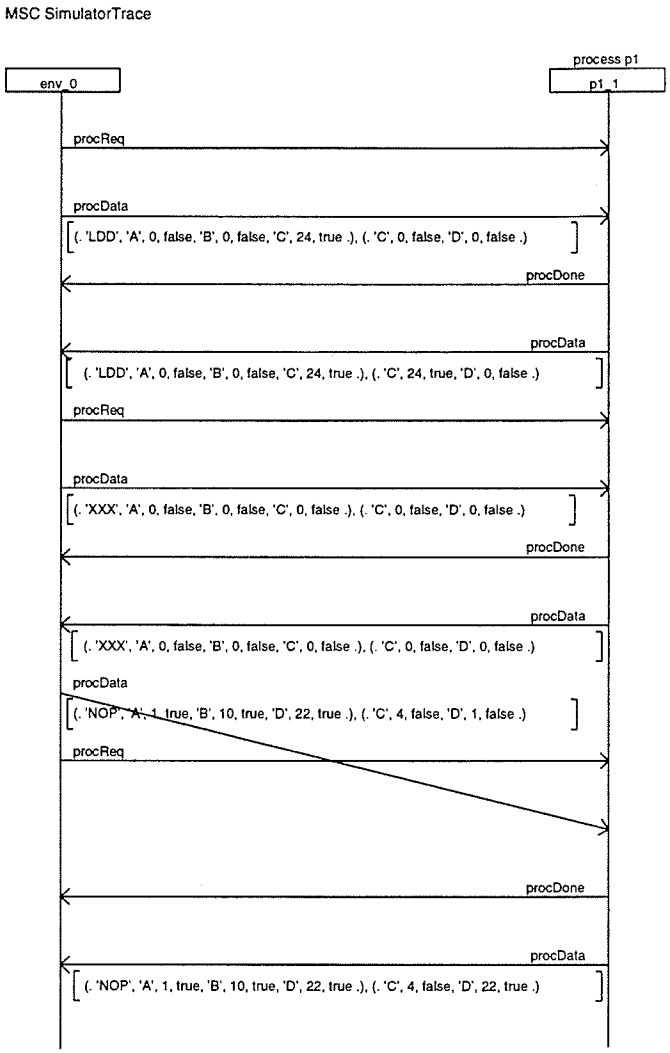


Figure 5.9: An MSC simulation trace of the processor

The next traces show that the execution of an unknown opcode ("XXX") resulted in untouched instruction and result registers. This is a feature of our design in which we can have different processors with different instruction sets. If the processor does not recognize the opcode, it will do nothing. The last traces show a situation in which a *procData* signal arrived before the *procReq* signal. This situation is possible because we use *two-phase* handshaking, and there is the possibility of a long delay in the *route* or *channel* assigned to the *procReq* signal. This is the purpose of using

the *save symbol* in the process type *PTGeneral*, otherwise the *procData* signal would be consumed and discarded. The system would then be deadlocked. This particular example also shows the delay-insensitive operation of the CFPP, in which the system still operates correctly, regardless the delays of the channels.

5.3.3 Stage Block Specification

The *stage block* is the key to the pipelining operation. It exchanges information with its neighboring blocks through its *data* and *control* channels. It accepts an instruction from the lower stage and passes the instruction to the upper stage. It also accepts a result from the upper stage and passes the result to the lower stage. One requirement is needed to guarantee the correctness of the design: "*The stage block should not pass an instruction to the upper stage and receive the result from the upper stage at the same time.*" If this requirement is violated, an instruction and a result may miss each other in a stage and therefore they will not go to a processor to be executed.

Each *stage* block is of type *SType*, which is shown in Figure 5.10. It consists of a process called *stagep2*. It was decided to have one process to handle traffic from both the upper and lower stages. Another possible approach would be to have two processes. Each process would be responsible for one flow direction, and the two processes would communicate with each other. This approach is more complex than the one-process approach. Unlike in the description of block type *PType*, we do not use a process type to describe block type *SType*, but rather describe the process *stagep2* directly (shown in Figure 5.12). Here, we are not using the SDL-92 extensions.

Originally, the *stagep2* algorithm (EFSM) handled the instruction and result flows symmetrically. There is a possible deadlock condition, in which two neighboring stages are waiting for each other. One stage is waiting for an instruction, while the

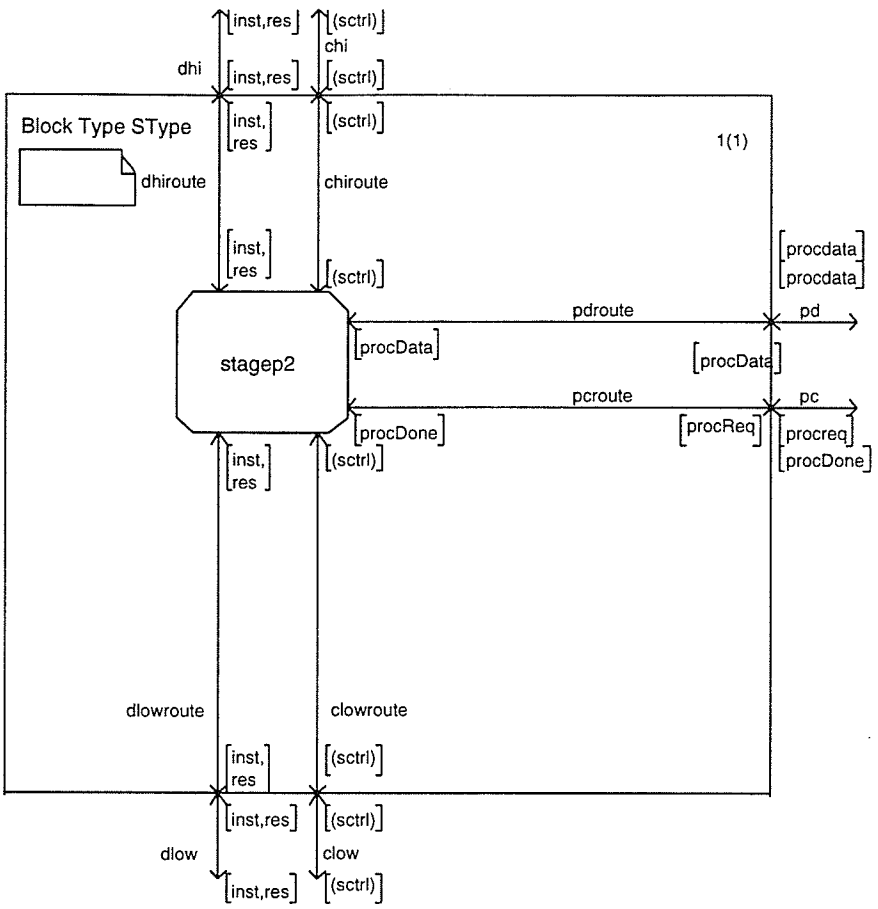


Figure 5.10: Description of block type SType

other one is waiting for a result. An informal description of this scenario is as follows. (See Figure 5.11.) Stage i receives an *instruction* from a lower stage while its *result register* is still empty, and stage $(i+1)$ receives a *result* from an upper stage while its *instruction register* is empty. In response, stage i sends a request to send an instruction (by sending an *instReq* signal) to stage $(i+1)$. While this request signal is on its way, stage $(i+1)$ sends a request to send a result (by sending a *resReq* signal) to stage i. Signal *instReq* arrives at stage $(i+1)$. Since stage$(i+1)$ has sent a *resReq* signal but did not receive any responses from stage i, it decides to accept an instruction by sending a *readyforInst* signal and it waits for the instruction. Stage i

received a *resReq* signal and since there is nothing happening (while the *readyforInst* signal is on its way), it decides to accept the result by sending a *readyforResult* signal and it waits for the result. Under this scenario, the system deadlocks! We found this condition through an MSC simulation by simultaneously sending (from the environment) an *instruction* to the "stage1" block and a result to the "stage3" block. Another deadlock condition is also possible if we let "stage 1" and "stage 3" simultaneously send an instruction and a result respectively. They are both waiting for the acknowledge signals. This scenario also does not satisfy our requirement described earlier, i.e. an instruction and a result may not miss each other.

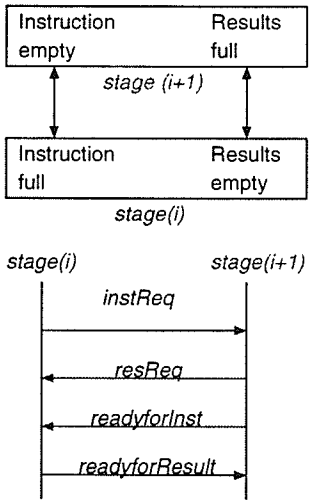


Figure 5.11: A deadlock situation

Our solution to this problem is to handle instruction and result flows asymmetrically. The complete description is presented in Figure 5.12 to Figure 5.15. If a stage has a full *instruction register* but an empty *result register*, it will wait for *readyforInst* or *resReq* signals from the upper stage. If it receives the *resReq* signal first, it will try to receive the *result* value and it will suspend other operations (by using the save symbol). This algorithm is shown on "*page 2(4)*" of *stagep2* in Figure 5.13. In essence, this algorithm tends to flow results faster than instructions. In [154], they

suggest operating the CFPP the other way around, i.e. to flow instructions as fast as possible. Further simulations are required to evaluate the performance of this case. Changing the "bias" – to flow instructions faster than results, or the inverse – would not be difficult.

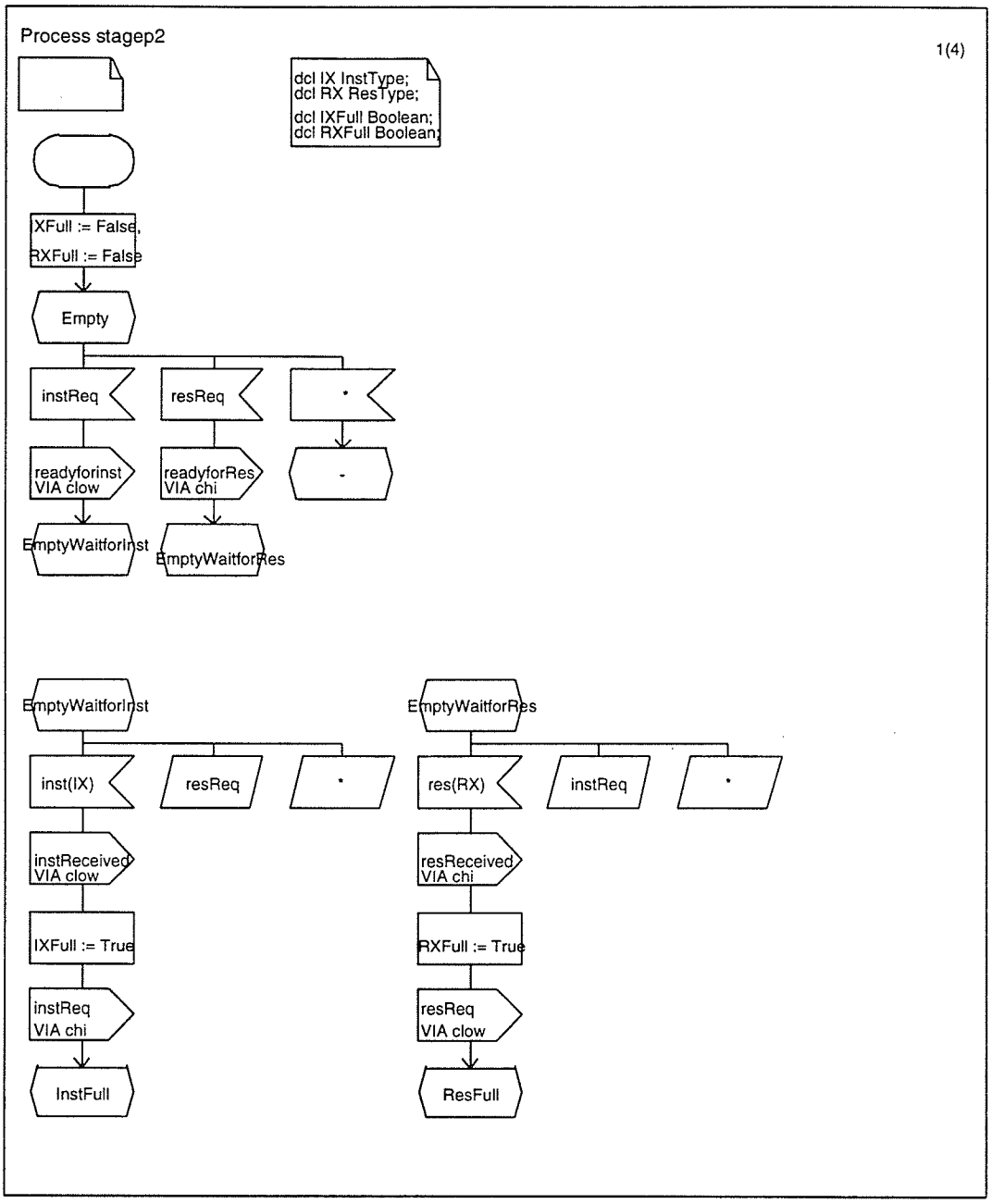


Figure 5.12: Process stagep2 - page 1

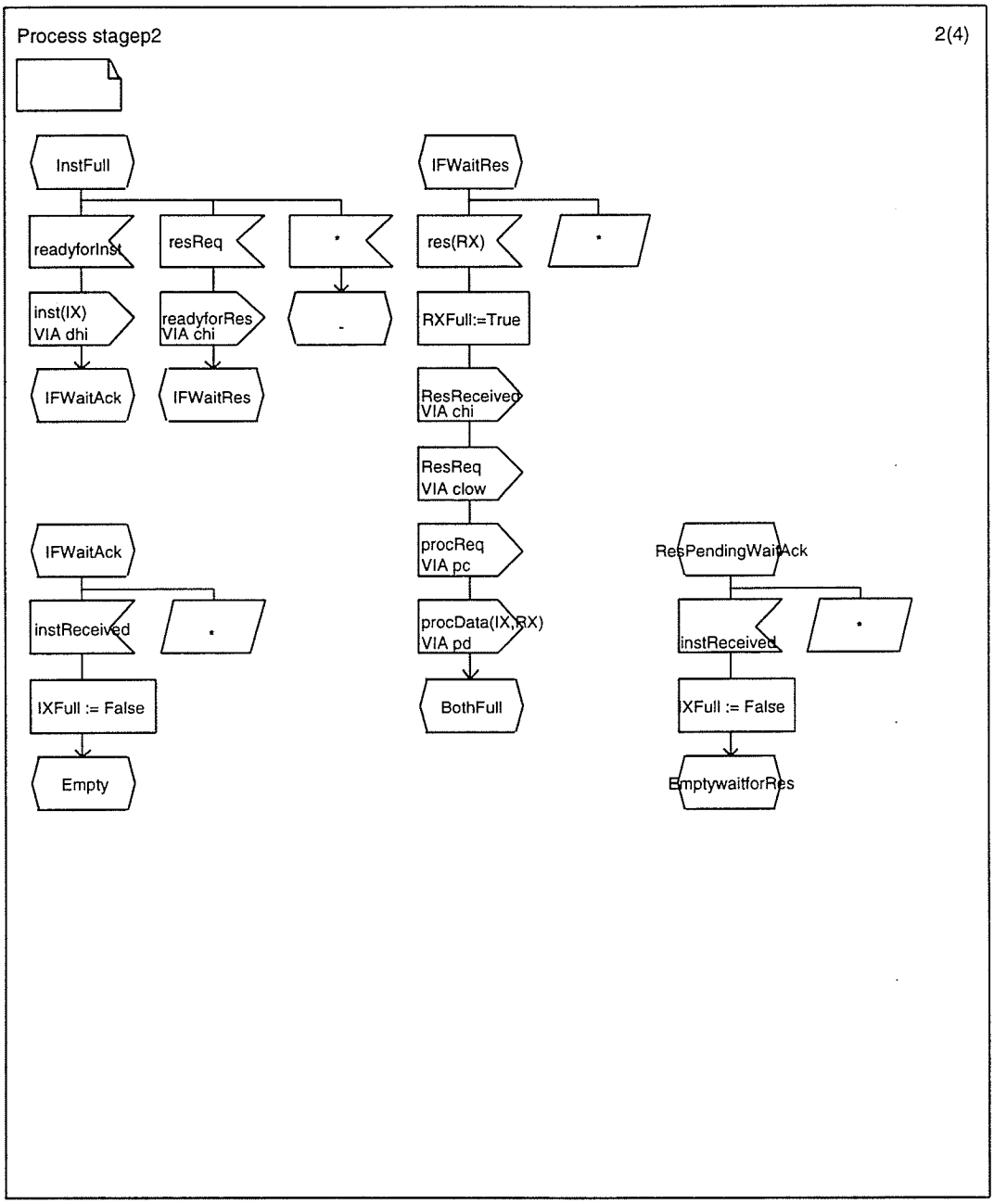


Figure 5.13: Process stagep2 - page 2

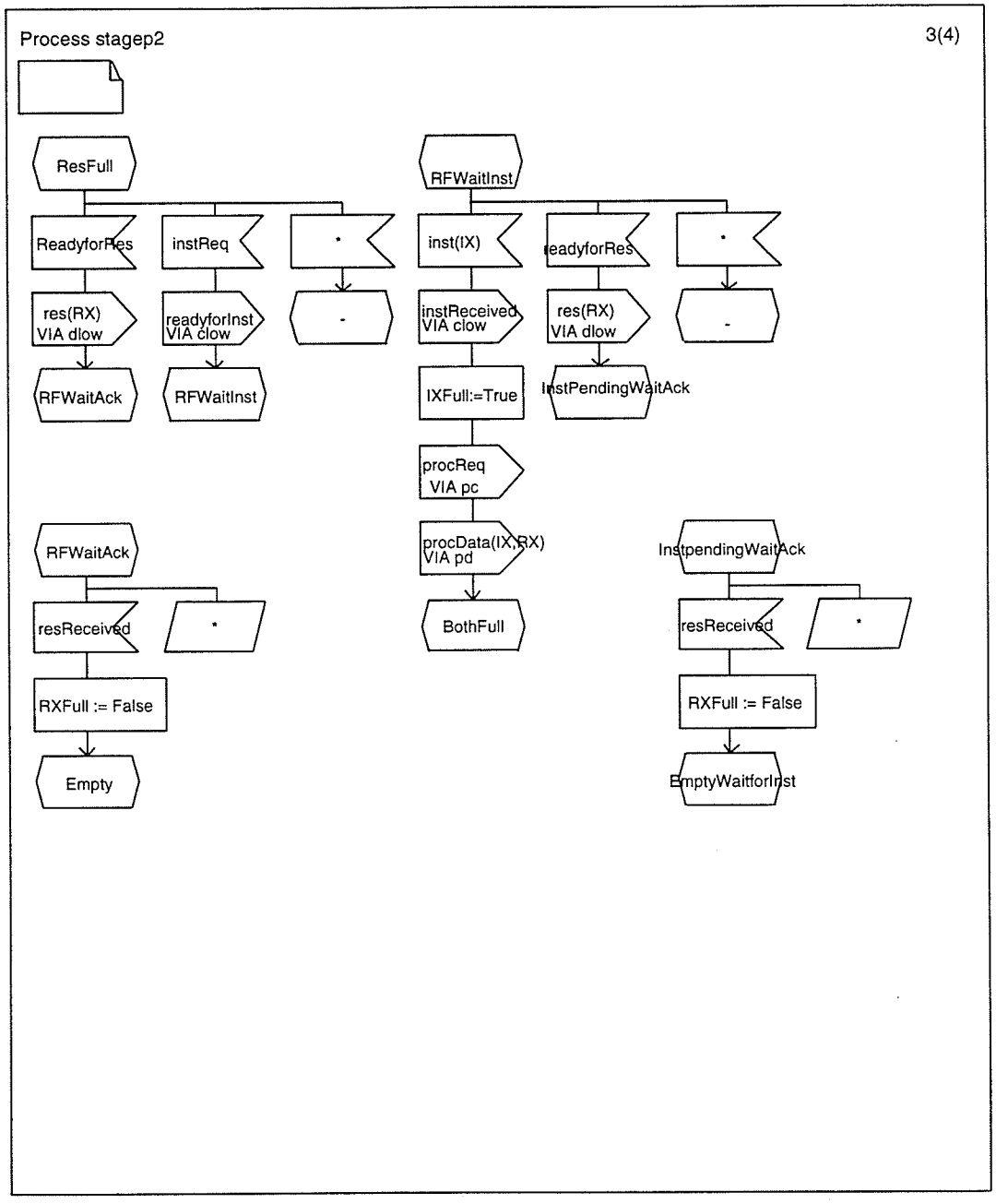


Figure 5.14: Process stagep2 - page 3

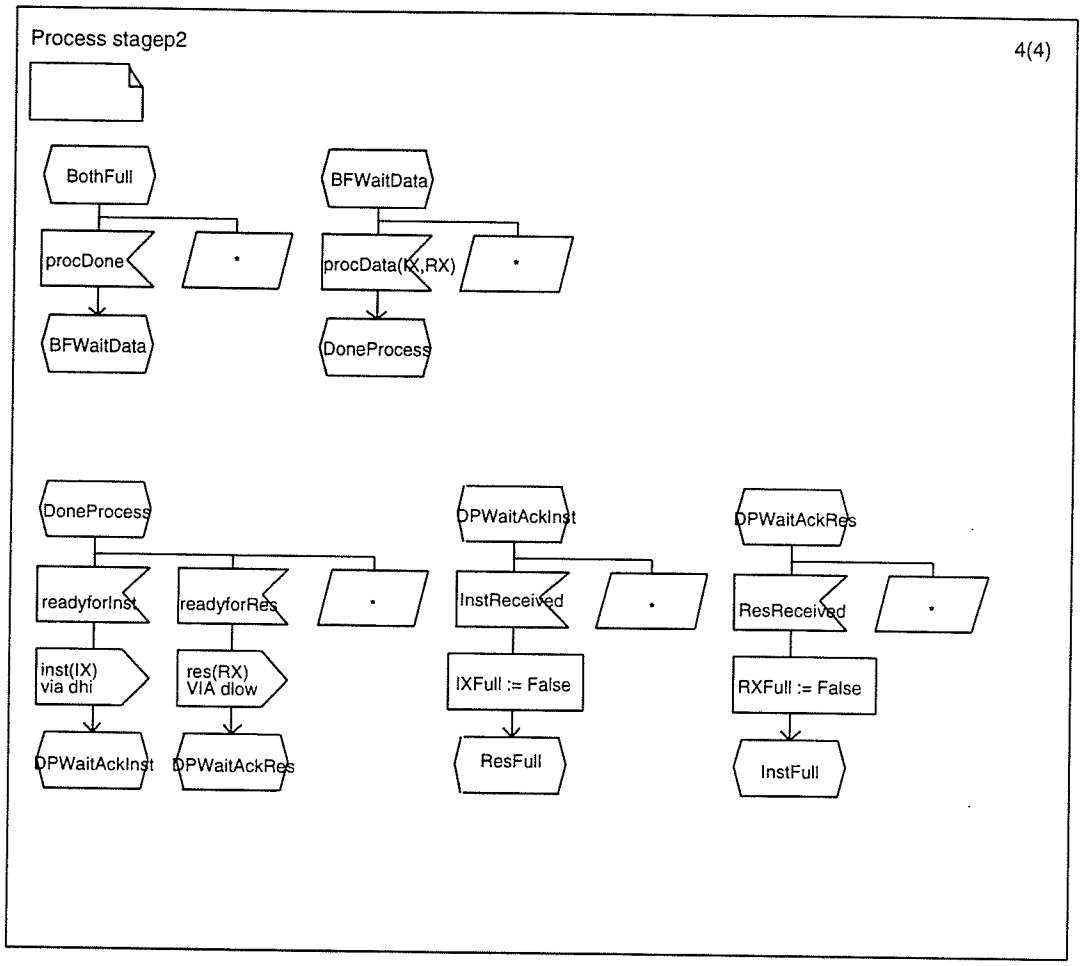


Figure 5.15: Process stagep2 - page 4

5.4 Validation

The SDT tool has a validation tool, called "SDT Validator", which can be used to find inconsistencies and problems. SDT Validator implements a state space exploration technique based on the same algorithm used in SPIN. Three types of automatic state space explorations are available: *Bit state exploration* (based on Holzmann's bit state algorithm [73]), *Random walk*, and *exhaustive exploration.*

One of the techniques used in validating a large system is to decompose the system into several smaller systems, hence reducing the size of the state space. We validated the processor block, which is a block type `PType`. We created a stand-alone system which consists of a block of block type `PType`. This block has a small number of signals: "*procData*", "*procReq*", and "*procDone*". Thus, the state space should be well within the SDT Validator capabilities.

The SDT Validator automatically generates test values that will be used as possible signals coming from the environment during validations. However, the automatic test value generation cannot generate suitable test values for our system. The signal "*procData*" has parameters of type "*InstType*" and "*ResType,*" two new types that we defined in the system level. These types have fields of type character (e.g. "*Result1*"). By default, the automatic test value generation produces an 'a' for the *Character* data type. Thus, test values for other possible accumulators (such as 'B' and 'C') cannot be generated automatically. Similarly, for the *Charstring* data type, the default value generated by the automatic test value generation is the word "test." We need to generate the words "LDD" and "NOP."

The SDT tool has a coverage viewer tool which can be used to measure how effective the validation tool is in validating the system. It measures the percentage of paths or states which have been visited. The tool has a graphical user interface to

investigate parts which have not been visited. The validation using the *bit state exploration* technique, and the values generated automatically produced 42% coverage. This means that only part of the design was exercised at least once. The other parts were not exercised at all. The other techniques, *random walk* and *exhaustive exploration*, produced similar results. The procedure *ProcessInst* had not been exercised completely due to the lack of good test patterns.

By supplying a set of selected test values, the validation achieved 100% coverage. The test values that we used are:

```
procData((. 'NOP', 'A', 0, true, 'B', 0, true, 'C', 2, false .),
   (. 'A', 55, true, 'A', 55, true, .))
procData((. 'NOP', 'A', 0, true, 'B', 1, true, 'C', 2, true .),
   (. 'C', 0, false, 'A', 0, false .))
procData((. 'NOP', 'A', 0, true, 'B', 1, true, 'C', 2, true .),
   (. 'A', 0, false, 'C', 0, false .))
procData((. 'NOP', 'A', 0, true, 'B', 1, true, 'C', 2, true .),
   (. 'A', 0, false, 'B', 0, false .))
procData((. 'LDD', 'A', 0, true, 'B', 1, true, 'C', 55, true .),
   (. 'A', 0, false, 'C', 0, false .))
procData((. 'LDD', 'A', 0, true, 'B', 1, true, 'C', 55, true .),
   (. 'A', 0, false, 'B', 0, false .))
procData((. 'LDD', 'A', 0, true, 'B', 1, true, 'C', 55, true .),
   (. 'C', 0, false, 'B', 0, false .))
```

The above test patterns were obtained by inspecting the algorithm used in the procedure *ProcessInst*, as shown in Figure 5.8. We guaranteed that each path will be exercised at least once. This method is acceptable for small systems, but it is inappropriate for large and complex systems. In complex systems, we may not be able to find a set of test values that will give 100% coverage. Attempts to provide all possible test values may also fail. For example, an exhaustive test value generation for our "*procData*" signal yields 98304 possible test patterns! (The program to generate the patterns is listed in Appendix D.) For more complex systems, the number grows exponentially.

Some statistics from the validation are shown below.

```
** Starting bit state exploration **
Search depth     : 100
Hash table size : 1000000 bytes

** Bit state exploration statistics **
No of reports: 0.
Generated states: 340.
Truncated paths: 0.
Unique system states: 196.
Size of hash table: 8000000 (1000000 bytes)
No of bits set in hash table: 392
Collision risk: 0 %
Max depth: 52
Current depth: -1
Min state size: 108
Max state size: 308
Symbol coverage : 100.00
```

5.5 Verification

One of the tools available from the SDT tool is a verification tool, which is actually based on the validation tool. The verification tool can be used to verify a design against an MSC diagram. The MSC serves as the specification of the design similar to a timing diagram in a conventional circuit design. MSCs can take different levels of abstraction. That is, we can hide some internal signals if our interest is mainly in monitoring the input and output signals. The verification tool takes an SDL system and exercises it to determine whether the system meets or violates a given MSC specification.

In this section we describe the verification of the processor (*PType* block) against a partial specification, which is illustrated in Figure 5.16. This specification says that the "LDD" command loads a value to an accumulator, i.e. given the following message

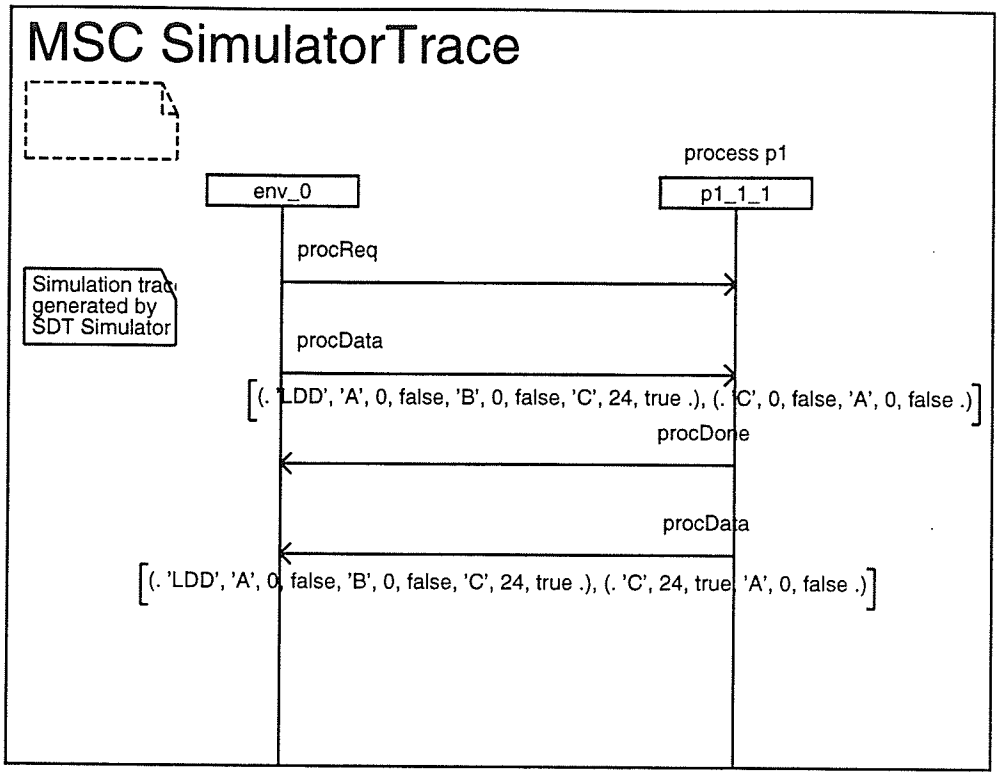


Figure 5.16: An MSC specification of block type PType

```
(. 'LDD', 'A', 0, false, 'B', 0, false, 'C', 24, true .),
(. 'C', 0, false, 'A', 0, false .)
```

the processor should set the value of accumulator "*C*" to '24' and should produce the following output

```
(. 'LDD', 'A', 0, false, 'B', 0, false, 'C', 24, true .),
(. 'C', 24, true, 'A', 0, false .)
```

Notice that the value of "*C*" is changed to '24' and its flag is set to TRUE.

A simple system was constructed based on the *PType* block type, and the MSC was verified to be correct with the SDT verification tool. The result of the verification is another MSC (shown in Figure 5.17). The verification session is listed below.

```
Welcome to SDT VALIDATOR
```

```
Command : verify-msc /home/unix/brahardj/SDT/validate-CFPP/spec1.msc

MSC SimulatorTrace loaded.
Root of behaviour tree set to current system state
Reports cleared
Bit state exploration started.

** Bit state exploration statistics **
No of reports: 1.
Generated states: 6.
Truncated paths: 0.
Unique system states: 5.
Size of hash table: 8000000 (1000000 bytes)
No of bits set in hash table: 10
Collision risk: 0 %
Max depth: 4
Current depth: 4
Min state size: 132
Max state size: 320
Symbol coverage :  57.89

** MSC SimulatorTrace verified **
```

The MSC specification we have is a partial specification, meaning that it does not describe all possible combinations. Thus, we only verified a partial system. It should be understood to mean that the verification tool only verifies a design against a given specification. It is also possible that the specification itself is faulty (i.e. it is not what we want) or incomplete.

The verification tool uses the same state-space search algorithm used in SPIN [73]. An advantage of this state-space exploration approach is that it can be automated, whereas a formal proof with mathematical logic would require more user interactions. This state-space exploration approach may not be suitable for applications that require reasoning or generalization. For example, it is difficult to create a complete MSC specification to describe the property of an "addition of two integers."

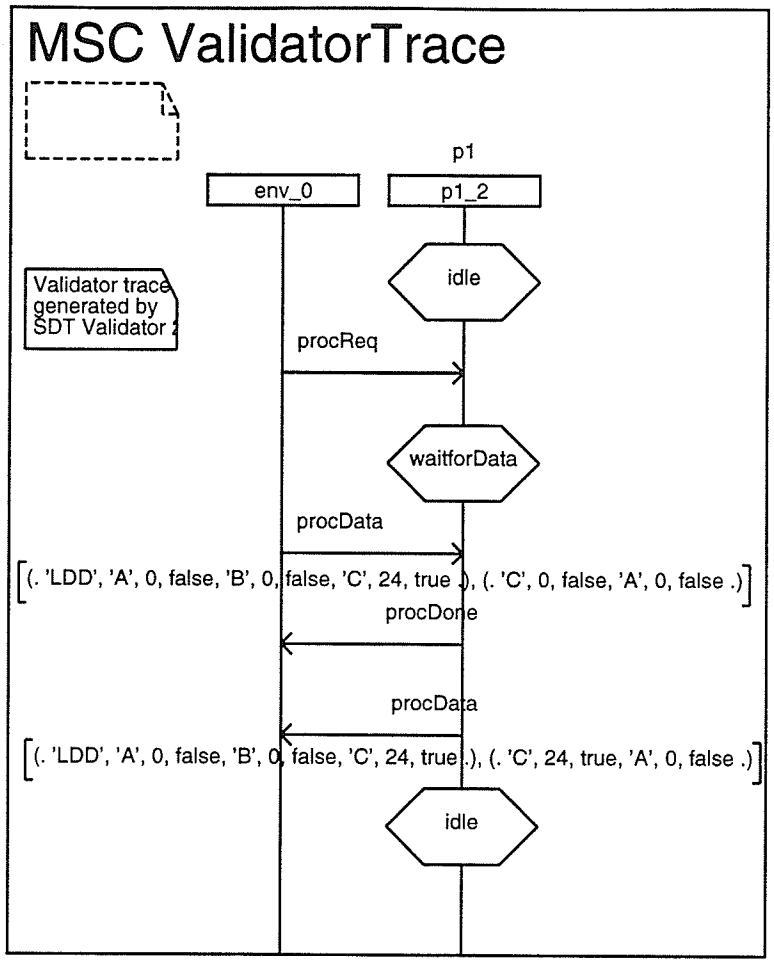


Figure 5.17: An MSC verification of block type PType

5.6 Discussion

We have presented a preliminary attempt to design a Counterflow Pipeline Processor in SDL-92. In this section, we discuss our experience with SDL and SDT, and provide some comments regarding the CFPP itself.

The use of SDL allows designers to spend more time in the design itself. The availability of a graphical representation (SDL-GR) makes it easier for beginners and even advanced users to use the SDL language. However, in some cases, such as the use of repetitive descriptions, the use of textual representation (SDL-PR) may be

better, as the textual representation also provides portability.

The *Abstract Data Type* (ADT) used in SDL allows the designer to concentrate on the design, rather than on the low-level bits and bytes. By doing so, one can restrain oneself from dealing with implementation details. The ADT also makes it possible to incorporate not only the control signal, but also the data part in the design.

Overall, SDT is an excellent design tool, even though there are still some bugs. For example, the MSC editor has some problems in drawing certain signals. It took us quite a while to figure out why certain signals were not drawn in the diagram. Originally, we thought that our description was erroneous. After inspecting SDT validator log file, it was shown that certain signals were not drawn, even though they actually existed in the system. Error messages were logged in the log file, indicating SDT difficulties in drawing long text messages.

MSCs provide a quick visual inspection of causal relationships. By looking at an *MSC trace*, one can quickly see if a signal is not consumed explicitly (indicated by an asterisk in the MSC trace) and can act upon it. Unfortunately, it is cumbersome to inspect an MSC if we have "too many" trace instances. In one of the simulations we have three stage blocks and three processor blocks. Even with this small number of instances, it is already difficult to find out the state of all processes. Fortunately, the SDT tool provides facilities to filter or display a range of instances[4].

After our design is completed in SDL, it can be translated to a VHDL description for hardware realization and simulations. This translation is discussed in [137]. The advantage of taking this route is that hardware synthesis from VHDL is a mature field. There are several industrial-strength synthesis tools (such as *Synopsys* and *Mentor Graphics*) that can take VHDL descriptions and turn them into circuits. Translations from SDL to C or C++ code which can be used to generate stand-alone software are

[4]In the newer release of the SDT tool, states can be displayed in MSC traces.

available.

The Counterflow Pipeline Processor is a unique computer architecture. It is still in its infancy, hence there have not been many studies to evaluate its advantages. Its regular structure makes it easier to be constructed automatically. Its local communication scheme potentially can make the system faster and more manageable than many traditional architectures.

5.7 Summary of Chapter 5

In this chapter, the use of SDL in the design and analysis of a Counterflow Pipeline Processor (CFPP) has been presented, as well as a novel asynchronous design of CFPP and its protocol. The design is easy to modify and extend. It is also amenable to formal verification. Validations and partial verifications of the design were done through a state-space exploration technique with the SDT Validator. The result is by no means restricted to the SDT tool, as other SDL verification tools can also be used. The state-space exploration algorithm used in the SDT Validator is the same algorithm used in the SPIN tool, which is used in Chapter 4.

The graphical nature of SDL makes it easier for system designers to master the SDL language. The availability of industrial-strength tools plays an important role in the acceptance of formal verification in the industry. These are the strengths of the SDL approach as compared with the use of PROMELA in the designing of asynchronous systems.

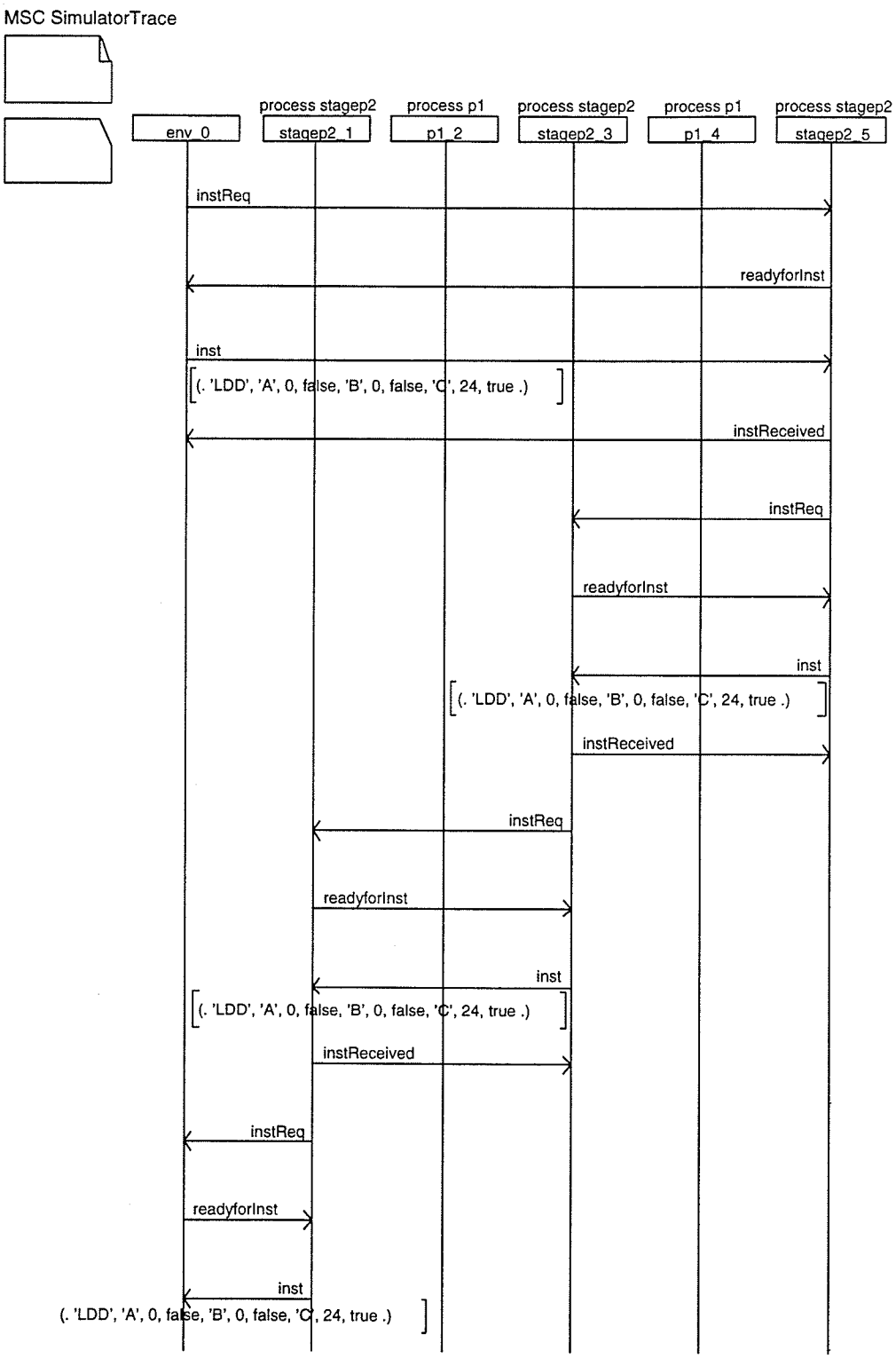


Figure 5.18: An MSC simulation trace of the CFPP

Chapter 6

Proving Asynchronous Circuits

In Chapter 3, "*Overview of Formal Methods for Hardware Design*," we briefly touched on proof-based formal verification methods. Verifications based on the state-space exploration method have been presented in Chapter 4 and Chapter 5. In this chapter, we use a proof-based method to reason about asynchronous circuits.

To aid in carrying out the proof, a theorem prover, or theorem proving environment is usually used. The theorem prover is used to prevent one from making errors. The theorem prover is not an automatic tool that can solve problems by itself. It still requires guidance and direction from a person. This human interaction is the main disadvantage of the proof-based method, as there are not many practitioners who are familiar with such methodology and tools. Proof-based verifications may not be effective for certain applications as the formal proofs can be long and complicated.

Although theorem provers have been used in hardware designs, to our knowledge they have not been extensively applied to asynchronous circuits. The applications of theorem provers to reason about asynchronous circuits were presented in [89], (they actually add delay elements to find hazards in their circuits), and in [55]. Barros and Johnson in [4] use first-order predicate logic to describe some classes of commonly used

asynchronous circuits. Recently, Kishinevsky and Staunstrup in [85] mechanically verified various speed-independent circuits with the Larch Prover [56].

In this chapter, higher-order logic has been selected as the specification language. It has been selected due to its expressiveness and the availability of tools to manipulate higher-order logic such as HOL [65], Isabelle [129], PVS [41, 40], and Voss [150]

Higher-order logic can be used to express several logics (e.g. temporal and modal logics) effectively [64]. Several formalisms (e.g. temporal logic, CSP, Z, and UNITY) which have been used to specify concurrent systems have been embedded into HOL. They are discussed in [13, 24, 135].

Embedding *Hardware Description Languages* (HDLs), such as ELLA, SILAGE, and VHDL in HOL is presented in [12]. Their experiences show that the biggest problem in reasoning about HDLs is to define the formal semantics of the HDLs, as the formal semantics were not part of their original definitions. Based on these works, reasoning about asynchronous circuits with higher-order logic is a viable approach.

6.1 Reasoning about C-elements

In Section 4.2 we presented a hazardous C-element and showed the presence of a hazard with PROMELA and SPIN. The state-space exploration technique was used to verify the circuit. In [131], we attempted to reason about the same C-element circuit by using *Real-Time CSP* (RTCSP) and higher-order logic.

RTCSP extends CSP by providing time-critical constructs for modeling timeouts, delays, and parallel processing. RTCSP has been mechanized in HOL. The syntax of RTCSP is listed in Appendix F. In [131] we show, by using HOL, that the firing of gate AND1 must be delayed so that both gate AND1 and OR1 can be fired simultaneously.

In essence, in this approach we reason about *Timed Asynchronous Circuits*, by

adding delay properties to components and wires. Although timed asynchronous circuits may be more practical, this approach does not work with pure delay-insensitive asynchronous circuits, in which the delays do not affect the correctness of the operation of a design. Our main interest is in the verification of pure delay-insensitive asynchronous circuits.

6.2 Specifications of Delay-Insensitive Components

We have attempted to find a generic higher-order specification for *delay-insensitive* circuits. To simplify the analysis and discussion, we have concentrated on a simple element: a delay-insensitive delay element (`Didel`), which delays its input by an unknown delay value. Although `Didel` is a simple element, it is an important element since we can construct a larger set of components by adding the `Didel` component to the inputs or outputs of ideal gates. The placement of the `Didel` component depends on the delay model of the circuit. If we attach `Didel` components at the input ports of ideal gates, we will have delay-insensitive circuits. If we attach `Didel` components at the output port of ideal gates, we will have *quasi delay-insensitive* circuits, i.e. wire forks have the same delay.

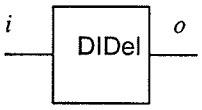


Figure 6.1: Didel, a delay-insensitive delay element

6.2.1 Straightforward specification

Adapting the notation used in [109], a straightforward higher-order specification of the `Didel` component is shown below.

$$\vdash \text{Didel}(i, o) = \forall t.\ o(t + t') = i\ t \tag{6.1}$$

This specification states that the value of output o is equal to the value of input i delayed by t' discrete time. The value of t' is unknown beforehand.

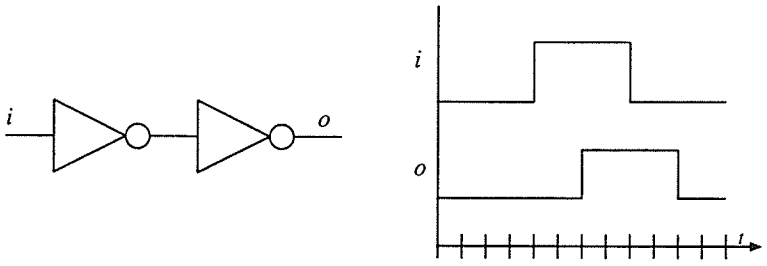


Figure 6.2: Two inverter (TwoInv) circuit and its timing diagram

This specification can be satisfied by a circuit ("`TwoInv`") which consists of two inverters in series, where each inverter delays its output by one unit time. The specifications of the inverter (`Inv`) and `TwoInv` are shown below.

$$\vdash \text{Inv}(i, o) = \forall t.\ o(t + 1) = \neg i\ t \tag{6.2}$$

$$\vdash \text{TwoInv}(i, x, o) = \text{Inv}(i, x) \wedge \text{Inv}(x, o) \tag{6.3}$$

The composition of the two inverters is based on the 'hiding' concept discussed in [109]. We hide the internal signal 'x'. The verification becomes proving the following theorem. (The proof is in Appendix G.)

$$\forall i\ o.\ \text{Didel}(i, o) \equiv \exists x.\ \text{TwoInv}(i, x, o) \tag{6.4}$$

The specification in (6.1) has two flaws. First, the delay t' is fixed for rising and falling transitions. In real circuits, rise time and fall time are usually different. A

possible solution to this flaw is to rewrite the specification by splitting it into fall and rise specifications. However, we are still left with a fixed delay for rising transitions and another fixed delay for falling transitions. In real circuits, two inverters may have different rise times.

The delay modeled in this specification is called *Fixed Ideal Delays* (FID) [20], i.e. the delay t' is fixed. However, rise and fall delays in real circuits may be functions of time, temperature, load, and age.

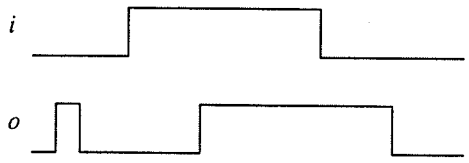


Figure 6.3: An incorrect input-output timing diagram of a Didel

The second flaw is that the specification (6.1) is satisfied by an incorrect implementation which has the input-output timing diagram shown in Figure 6.3. That is, a spike existed before the implementation stabilizes. This condition can occur after a *reset* or a *restart* period, but before an input is applied. If the output o is connected to an input of another device, the spike will be propagated. In the specification, we specify the output o after time t'. The output o before time t' is not specified.

6.2.2 Partial-Order Specification

By looking at the ordering of events (rising or falling transitions), there exists another possible means of specification for delay-insensitive circuits. Since we do not know the exact delay of each component, temporal abstraction is performed. The specification is done at a higher level of temporal abstraction, i.e. by monitoring the sequence of events. The partial-order specification of the `Didel` component is shown below.

$$\vdash \text{Didel}(i, o) = (\text{Timeof (Rise } o) \; n \geq \text{Timeof (Rise } i) \; n) \wedge \\ (\text{Timeof (Fall } o) \; n \geq \text{Timeof (Fall } i) \; n) \tag{6.5}$$

where the term 'Timeof P n' denotes the point at a concrete (real) time in which predicate P is true for the n-th time. (For a more rigorous justification of the use of `Timeof()`, we refer the reader to [108, 109].) `Rise` and `Fall` are defined below.

$$\vdash \text{Rise } ck \; t = \neg ck(t) \wedge ck(t+1) \tag{6.6}$$

$$\vdash \text{Fall } ck \; t = ck(t) \wedge \neg ck(t+1) \tag{6.7}$$

The partial-order specification states that the concrete time when the n-th event of output "o" occurs must be greater than the concrete time when the n-th event at input "i" occurs. In other words, an event at the output o must have a corresponding (earlier) event at the input i. Oscillations, spikes, or hazards at the output o are not allowed since they do not have corresponding input events.

This partial-order specification is stronger as compared with the straightforward specification. Only hazard-free implementations are allowed. Thus, this specification is preferred as compared with the straightforward specification.

6.3 Summary of Chapter 6

We have presented some initial work in reasoning about asynchronous circuits with higher-order logic. Two possible higher-order specifications of a delay-insensitive de-

lay element were presented. The first specification, the straight-forward specification, has two flaws, whereas the partial order specification does not have those flaws.

As noted earlier, *delay-insensitive* and *quasi delay-insensitive* asynchronous circuits can be decomposed into ideal gates and `Didel` components. This study is an initial step towards the verification of a larger set of delay-insensitive and quasi delay-insensitive circuits.

Chapter 7

Conclusions and Future Research

7.1 Conclusions

In this thesis we have presented various aspects of the design and formal verification of asynchronous circuits and systems. The use of higher level or system level methodologies is suggested. Specifically, the use of protocol engineering in the design, validation and verification of asynchronous systems with PROMELA and SDL is identified as a suitable methodology.

Formal verification of asynchronous systems was demonstrated by providing several verification runs. Several asynchronous circuits, and an asynchronous system, were modeled in PROMELA and SDL, and they were verified with the SPIN and the SDT tool respectively. The verifications were done using the state-space exploration method.

Some initial work on proof-based verification has been presented in Chapter 6. Real-Time CSP (RTCSP) and higher-order logic have been used to model asynchronous circuits.

7.2 Contributions of this Thesis

This thesis is one of the first attempts to identify system level design methodologies specifically oriented to asynchronous systems, with an emphasis on the use of formal methods as an integral part of the methodology. This is accomplished by considering various levels of abstractions from asynchronous circuits to asynchronous systems. With different levels of abstraction, various formal methods have been identified and their usage was illustrated through design examples. As opposed to reinventing tools within an asynchronous system design methodology, existing ones were adapted for use in the design of asynchronous systems. Specific instances include: i) one of the first attempts to use PROMELA and SPIN to specify and verify asynchronous building blocks such as C-elements; and ii) one of the first in using SDL to design and verify the Sproull's *Counterflow Pipeline Processor*. The methodology and design we presented will precede any commercial implementation of these types of processors.

Specific contributions of this thesis include:

1. A comprehensive overview of design methodologies for asynchronous hardware (in Chapter 2).

2. The identification of an integrated asynchronous system design methodology using protocol engineering with PROMELA and SDL (in Chapter 4 and Chapter 5).

3. Verifications of asynchronous circuits and systems through state-space exploration techniques with protocol validation tools. Two verification approaches are illustrated: by modeling several asynchronous circuits and an asynchronous system in PROMELA and SDL, and by using the SPIN and the SDT tool respectively to verify the circuits and system. The verifications have been presented

in Chapter 4 and Chapter 5.

4. Programming techniques to model *speed-independence*, *fundamental mode*, and *input-output mode* asynchronous circuits in PROMELA.

5. A novel design of a *Counterflow Pipeline Processor* in SDL, which has been presented in Chapter 5.

6. A preliminary attempt to reason about asynchronous circuits through proof-based verification by using *Real Time CSP* (RTCSP) and higher-order logic (in Chapter 6).

7.3 Future Research

Future research which can be done is listed below.

1. Automation of the methods which have been presented in this thesis. Automation of verification is desirable, especially in verifying large systems.

2. An integration of the methods with some VLSI CAD tools. There is an interest in integrating formal verification methods with mature VLSI CAD tools to provide better quality circuits and an increase in productivity.

3. Further research in proof-based verification. The proof-based approach has a potential advantage, in that it can be used to reason about infinite systems. However, it requires a great deal of user sophistication and interaction. A better understanding of this approach can be achieved through education and by providing more automated and user-friendly tools.

Bibliography

[1] G. A. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, 1986.

[2] R. Andrew. An algorithm for eight-valued simulation. In *Proceedings of the International Symposium on Multiple-Valued Logic*, pages 273–280, 1986.

[3] G. Barrett. Formal methods applied to a floating-point number system. *IEEE Transactions on Software Engineering*, pages 611–621, May 1989.

[4] J. C. Barros and B. W. Johnson. Equivalence of the arbiter, the synchronizer, the latch, and the inertial delay. *IEEE Transactions on Computers*, pages 603–614, July 1983.

[5] H. G. Barrow. Proving the correctness of digital hardware designs. *VLSI Design*, pages 64–77, July 1984.

[6] H. G. Barrow. Verify: A program for proving correctness of digital hardware designs. *Artificial Intelligence*, 24:437–491, 1984.

[7] P. A. Beerel and T. H.-Y. Meng. Semi-modularity and self-diagnostic asynchronous control circuits. In *Advance Research in VLSI*, pages 103–117. MIT Press, 1991.

[8] F. Belina and D. Hogrefe. The CCITT specification and description language SDL. *Computer Networks and ISDN Systems*, (16):311–341, 1988.

[9] M. Ben-Ari. *Mathematical Logic for Computer Science.* Prentice-Hall, 1993.

[10] G. Birtwistle, Y. Liu, J. Aldwinckle, K. Stevens, and W. Yu. Case studies in asynchronous design: Part 1. AMM architecture, Part 2. a 4 stroke AMM. Draft, Dept. of Computer Science, University of Calgary, December 1993.

[11] G. V. Bochmann. Hardware specification with temporal logic: An example. *IEEE Transactions on Computers*, pages 223–231, March 1982.

[12] R. Boulton, A. Gordon, M. Gordon, J. Harrison, J. Herbert, and J. Van Tassel. Experience with embedding hardware description languages in HOL. In V. Stavridou, T. F. Melham, and R. T. Boute, editors, *Theorem Provers in Circuit Designs, Vol A-10, IFIP Transactions*, pages 129–156. North-Holland, Amsterdam, 1992. ftp:// ftp.cl.cam.ac.uk/ hvg/ papers/ EmbeddingPaper.ps.

[13] J. Bowen. Z and HOL. Unpublished manuscript. ftp:// ftp.cl.cam.ac.uk/ hvg/ papers/ zhol.ps.

[14] J. P. Bowen and M. G. Hinchey. Ten commandments of formal methods. Technical Report TR-350, Oxford University Computing Laboratory and University of Cambridge Computer Laboratory, 1994. http://www.cl.cam.ac.uk/users.

[15] G. M. Brown. Towards truly delay-insensitive circuit realizations of process algebras. In G. Jones and M. Sheeran, editors, *Designing Correct Circuits*, pages 120–131. Springer-Verlag, 1991. Workshops in Computing.

[16] M. C. Browne, E. M. Clarke, D. L. Dill, and B. Mishra. Automatic verification of sequential circuits using temporal logic. *IEEE Transactions on Computers*, C-35(12):1035–1044, December 1986.

[17] E. Brunvand. Part-R-Us. Technical report, Carnegie Mellon University, 1987.

[18] E. Brunvand. A cell set for self-timed design using Actel FPGAs. Technical Report UUCS-91-013, Dept. of Computer Science, University of Utah, August 1991.

[19] J. A. Brzozowski and J. C. Ebergen. On the delay-sensitivity of gate networks. *IEEE Transactions on Computers*, 41(11):1349–1360, November 1992.

[20] J. A. Brzozowski and C.-J. Seger. *Asynchronous Circuits*. Springer-Verlag, 1994.

[21] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Proceedings of the Fifth Annual Symposium on Logic in Computer Science*, June 1990. ftp://emc.cs.cmu.edu /pub/tape/SMC.ps.

[22] R. W. Butler and G. B. Finelli. The infeasibility of quantifying the reliability of life-critical real-time software. *IEEE Transactions on Software Engineering*, 19(1):3–12, January 1993.

[23] A. Camilleri, M. Gordon, and T. Melham. Hardware verification using higher-order logic. In D. Borrione, editor, *From HDL Descriptions to Guaranteed Correct Circuit Designs*. North-Holland, 1987.

[24] A. J. Camilleri. Mechanizing CSP trace theory in higher order logic. *IEEE Transactions on Software Engineering*, 16(9):993–1004, September 1990.

[25] P. Camurati, T. Margaria, and P. Prinetto. Use of the OTTER theorem prover for the formal verification of hardware. In G. Jones and M. Sheeran, editors, *Designing Correct Circuits*. Springer-Verlag, 1991.

[26] P. Camurati and P. Prinetto. Formal verification of hardware correctness: Introduction and survey of current research. *IEEE Computer*, pages 8–19, 1988.

[27] CCITT. *CCITT Blue Book, Vol. 10 - fasc. X.1, recommendation Z.100 - Functional Specification and Description Languages (SDL)*, 1989. Geneva.

[28] K. M. Chandy and J. Misra. *Parallel Program Design: A Foundation.* Addison-Wesley, 1988.

[29] T.-A. Chu. On the models for designing VLSI asynchronous digital systems. *INTEGRATION*, 4:99–113, 1986.

[30] T.-A. Chu. *Synthesis of Self-Timed VLSI Circuits from Graph-theoretic Specification.* PhD thesis, MIT, 1987.

[31] T.-A. Chu and C. K. C. Leung. Design of VLSI asynchronous FIFO queues for packet communication networks. In *Proceedings of the International Conference on Parallel Processing*, pages 397–400, 1986.

[32] W. A. Clark. Macromodular computer systems. In *AIFPS Conference Proceedings 1967 Spring Joint Computer Conference*, pages 335–336. Academic Press, April 1967.

[33] W. A. Clark and C. E. Molnar. Macromodular computer systems. In R. W. Stacy and B. D. Waxnab, editors, *Computers in Biomedical Research*, pages 45–85. Academic Press, 1974.

[34] E. M. Clarke and O. Grümberg. Reasearch on automatic verification of finite-state concurrent systems. *Ann. Rev. Comput. Sci.*, pages 269–290, 1987.

[35] B. Coates, A. Davis, and K. Stevens. The Post Office experience: Designing a large asynchronous chip. *Integration, the VLSI journal*, 15:341–366, 1993.

[36] B. Coates, A. Davis, and K. S. Stevens. Automatic synthesis of fast compact self-timed control circuits. In *Proceedings of the IFIP Working Conference on Design Methodologies*, pages 193–208, Manchester, 1993.

[37] A. Cohn. A proof of correctness of the VIPER microprocessor: the first level. In G. Birtwistle and P. A. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*. Kluwer, 1988.

[38] R. Comerford. How DEC developed Alpha. *IEEE Spectrum*, 29(7):26–31, July 1992.

[39] J. Cortadella, M. Kishinevsky, A. Kodratyev, L. Lavagno, and A. Yaokvlev. Petrify: a tool for manipulating concurrent specifications and synthesis of asynchronous controllers. Technical report, Departament d'Arquitectura de Computados, Universitat Politecnica de Catalunya, April 1996.

[40] D. Cyrluk. Microprocessor verification in PVS: A methodology and simple example. Technical Report CSL-93-12, Computer Science Laboratory, SRI International, 1994.

[41] D. Cyrluk, S. Rajan, N. Shankar, and M. K. Srivas. Effective theorem proving for hardware verification. In *2nd International Conference on Theorem Provers in Circuit Design*, 1994.

[42] M. E. Dean, T. E. Williams, and D. L. Dill. Efficient self-timing with level-encoded 2-phase dual-rail (LEDR). In C. H. Séquin, editor, *Advance Research in VLSI*, pages 55–70. MIT Press, Santa Cruz, 1991.

[43] H. M. Deitel. *An Introduction to Operating Systems.* Addison-Wesley, 1983.

[44] B. L. Di Vito, R. W. Butler, and J. L. Caldwell. Formal design and verification of a reliable computing platform for real-time control (phase 1 results). NASA technical memorandum 102716, NASA Langley Research, October 1990. file:// air16.larc.nasa.gov/ pub/fm/larc/ RCP-papers/rcp-phase-I.ps.

[45] D. L. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits.* MIT Press, 1988.

[46] D. L. Dill and E. M. Clarke. Automatic verification of asynchronous circuits using temporal logic. *IEE Proc. Pt. E*, 133(5):276–294, September 1986.

[47] J. C. Ebergen. A technique to delay-insensitive VLSI circuits. Technical Report CS-R8622, Centre for Mathematics and Computer Science, Department of Algorithmics and Architecture, Amsterdam, 1986.

[48] J. C. Ebergen. Parallel computations and delay-insensitive circuits. In G. Birtwistle, editor, *Proceedings of the IV Higer Order Workshop Banff 1990*, pages 85–104. Springer-Verlag, Banff, Alberta, Canada, Sept. 1990.

[49] J. C. Ebergen and S. Gingras. A verifier for network decompositions of command-based specifications. In *Proceedings of the 1993 Hawaii International Conference on Systems Sciences*, Jan. 1993.

[50] E. B. Eichelberger. Hazard detection in combinational and sequential switching circuits. *IBM Journal of Research and Development*, 9:90–99, December 1965.

[51] H. Eveking. Verification, synthesis, and correctness-preserving transformations-cooperative approaches to correct hardware design. In *From HDL Descriptions to Guaranteed Correct Circuit Designs*, pages 229–239. Elsevier Science Publishers B.V., 1987.

[52] O. Færgemand and A. Olsen. Introduction to SDL-92. *Computer Networks and ISDN Systems*, (26):1143–1167, 1994.

[53] G. Fantauzzi. An algebraic model for the analysis of logical circuits. *IEEE Transactions on Computers*, C-23(6):576–581, June 1974.

[54] M. Gamble, B. Rahardjo, and R. D. McLeod. Reconfigurable FPGA micropipeline. In *Proc. 1994 Canadian Workshop on Field Programmable Devices FPGA*, 1994.

[55] S. Garland, J. Guttag, and J. Staunstrup. Verification of VLSI circuits using LP. In G. J. Milne, editor, *The Fusion of Hardware Design and Verification.* Elsevier Science Publishers B.V., 1988.

[56] S. J. Garland and J. V. Guttag. A guide to LP, the Larch Prover. Technical report, Digital SRC, 1991.

[57] J. D. Garside. A CMOS VLSI implementation of an asynchronous ALU. In *Proceedings of the Manchester IFIP Working Conference on Asynchronous Design Methodologies*, 1993.

[58] S. M. German and Y. Wang. Formal verification of parameterized hardware designs. In *Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers*, pages 549–552, 1985.

[59] A. Ghosh, S. Devadas, and A. R. Newton. *Sequential Logic Testing and Verification.* Kluwer Academic Publishers, 1992.

[60] R. M. Goodman, K. A. Kramer, and A. J. McAuley. Exploiting the inherent fault tolerance of asynchronous arrays. In *Proceeding of the International Conference on Systolic Arrays*, pages 567–576, Killarney-Ireland, May 1989.

[61] G. Gopalakhrisnan, E. Brunvand, N. Michell, and S. M. Nowick. A correctness criterion for asynchronous circuit validation and optmization. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(11):1309–1318, November 1994.

[62] G. Gopalakrishnan and V. Akella. VLSI asynchronous systems: Specification and synthesis. *Microprocessors and Microsystems*, 16(10):517–527, 1992.

[63] M. Gordon. Why higher-order logic is a good formalism for specifying and verifying hardware. In G. Milne and P. A. Subrahmanyam, editors, *Formal Aspects of VLSI Design*, pages 153–177. Elsevier Science Publishers B.V., 1986.

[64] M. J. C. Gordon. Mechanizing programming logics in higher-order logic. In G. Birtwistle and P. Subrahmanyam, editors, *Current Trends in Hardware Verification and Automated Theorem Proving*, pages 387–439. Springer-Verlag, 1989. Also in Proc. Banff Workshop Hardware Verification, Banff, Canada, 1988.

[65] M. J. C. Gordon and T. F. Melham. *Introduction to HOL: A theorem proving environment for higher-order logic.* Cambridge University Press, 1993.

[66] A. Gupta. Formal hardware verification methods: A survey. In *Formal Methods in System Design*, volume 1. Kluwer Academic Publishers, October 1992.

[67] F. K. Hanna and N. Daeche. Specification and verification using higher order logic. In *Proceedings of the 7th International Symposium on Computer Hardware Description Languages and Applications*, pages 418–443, Tokyo, Japan, August 1985.

[68] F. K. Hanna and N. Daeche. Specification and verification of digital systems using higher-order predicate logic. *IEE Proc. Pt. E*, 133(5):242–254, September 1986.

[69] S. Hauck, S. Burns, G. Borriello, and C. Ebeling. An FPGA for implementing asynchronous circuits. Technical report, Dept. of Computer Science and Engineering, University of Washington, 1994.

[70] M. G. Hinchey and J. P. Bowen. To formalize or not to formalize. *IEEE Computer*, 29(4):18–19, April 1996.

[71] C. A. R. Hoare. Communicating sequential processes. *Communication of the ACM*, 21(8):666–677, August 1978.

[72] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985.

[73] G. J. Holzmann. *Design and Validation of Computer Protocols.* Prentice Hall Software Series. Prentice Hall, 1991.

[74] W. A. Hunt. FM8501: A verified microprocessor. In *IFIP WG 10.2 Workshop, From HDL Descriptions to Guaranteed Correct Circuit Design*, pages 85–114. North-Holland Publishing, 1986.

[75] IEEE. *IEEE Standard VHDL Language Reference Manual*, March 1988. IEEE Std 1086-1987.

[76] G. M. Jacobs. *Self-Timed Integrated Circuits for Digital Signal Processing.* PhD thesis, Electrical and Computer Science, University of California at Berkeley, December 1989.

[77] J. M. Johnson. *Theory and Application of Self-Timed Integrated Systems Using Ternary Logic Elements.* PhD thesis, Electrical and Computer Engineeing, University of California, Santa Barbara, December 1988.

[78] G. Jones. Sharp as a razor: A Queen's award for the computing laboratory. *Oxford Magazine*, Fourth Week, Trinity term(59), 1990. Also available, as ftp:// ftp.comlab.ox.ac.uk/ pub/Documents/ techpapers/ Geraint.Jones/ QATA-1-90.ps.Z.

[79] J. Joyce, J. Rushby, N. Shankar, R. Suaya, and F. von Henke. From formal verification to silicon compilation. In *Proceedings of the IEEE Compcon 1991*, pages 450–455, February 1991.

[80] J. J. Joyce. Verification and implementation of a microprocessor. In G. Birtwistle and P. A. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis.* Kluwer, 1988.

[81] G. M. Karam and R. J. A. Buhr. Starvation and critical race analyzers for Ada. *IEEE Transactions on Software Engineering*, 16(8):829–843, August 1990.

[82] K. Keutzer, L. Lavagno, and A. Sangiovanni-Vincentelli. Synthesis for testability technique for asynchronous circuits. In *ICCCAD 91*, pages 326–329, 1991.

[83] M. Kishinevsky. Hazard-free c-element. Personal (e-mail) communication (asynchronous @hohum.stanford.edu).

[84] M. Kishinevsky, A. Kondratyev, A. Taubin, and V. Varshavsky. *Concurrent Hardware: The Theory and Practice of Self-Timed Design.* Series in Parallel Computing. John Willey & Sons, 1993.

[85] M. Kishinevsky and J. Staunstrup. Mechanized verification of speed-independence. In R. Kumar and T. Kropf, editors, *Proceedings of the Second Conference on Theorem Provers in Circuit Design: Theory, Practice & Experience*, pages 229–247. FZI-Publication, 1994.

[86] W. H. F. J. Korver and I. M. Nedelchev. An asynchronous implementation of SCPP-A. Technical Report TR-CSRG-95-07, Computer Systems Research Dept. of Electronic & Electrical Engineering, University of Surrey, UK, July 1995.

[87] R. Kumar, T. Kropf, and K. Schneider. First steps towards automating hardware proofs in HOL. In M. Archer, J. Joyce, K. N. Levitt, and P. J. Windley, editors, *International Workshop on the HOL Theorem Proving System and Its Applications*, pages 190–193. IEEE Computer Society Press, Davis, California, August 1991.

[88] R. Kumar, T. Kropf, and K. Schneider. Integrating a first-order automatic prover in the HOL environment. In M. Archer, J. Joyce, K. N. Levitt, and P. J. Windley, editors, *International Workshop on the HOL Theorem Proving System and Its Applications*, pages 170–176. IEEE Computer Society Press, Davis, California, August 1991.

[89] R. Kumar, K. Schneider, and T. Kropf. Structuring and automating hardware proofs in higher-order theorem proving environment. In R. Brayton, M. Clarke,

and P. A. Subrahmanyam, editors, *International Journal of Formal System Design*, pages 165–230. Kluwer Academic Publishers, 1993.

[90] S. Y. Kung and R. J. Gal-Ezer. Synchronous versus asynchronous computation in Very Large Scale Integration (VLSI) array processors. *SPIE, Real Time Signal Processing V*, pages 53–65, 1982.

[91] L. Lamport. How to write a proof. Technical Report SRC-094, System Research Center, DEC, February 1993.

[92] T. W. S. Lee, M. R. GreenStreet, and C.-J. Seger. Automatic verification of asynchronous circuits. Technical Report 93-40, Computer Science, Univeristy of British Columbia, November 1993.

[93] L. F. Lind and J. C. C. Nelson. *Analysis and Design of Sequential Digital Systems.* Macmillan Press Ltd., 1977.

[94] Y. Liu. Reasoning about asynchronous design in CCS. Master's thesis, University of Calgary, 1992.

[95] D. E. Long. *Model Checking, Abstraction, and Compositional Verification.* PhD thesis, Carnegie Mellon University, 1993.

[96] D. MacKenzie. The fangs of the VIPER. *Nature*, 352:467–468, Aug. 1991.

[97] K. Maheswaran and V. Akella. Hazard-free implementation of the self-timed cell set for the Xilinx 4000 series FPGA. Technical report, Dept. of EE, University of California, Davis, 1994. ftp:// www.ece.ucdavis.edu/ pub/akella/Hazard.ps.Z.

[98] J. Markoff. Circuit flaw causes pentium chip to miscalculate. *New York Times*, Nov. 1994. 11/24/94.

[99] A. Marshall, B. Coates, and P. Siegel. The design of an asynchronous communications chip. *IEEE Design & Test*, June 1994.

[100] A. J. Martin. Compiling communicating processes into delay-insensitive VLSI circuits. *Distributing Computing*, pages 226–234, 1986.

[101] A. J. Martin. A synthesis method for self-timed VLSI circuits. *IEEE*, pages 224–229, 1986.

[102] A. J. Martin. The design of a delay-insensitive microprocessor: An example of circuit synthesis by program transformation. In M. Leeser and G. Brown, editors, *Hardware Specification, Verification and Synthesis: Mathematical Aspects*, pages 85–104. Springer-Verlag, 1989.

[103] A. J. Martin. Tomorrow's digital hardware will be asynchronous and verified. In *Proceedings of the IFIP Congress 1992: Information Processing 1992, Volume I.* Elsevier Science Publishers B.V., 1992.

[104] P. Mayo. Decision diagrams for ternary expressions. Master's thesis, Computer Science, University of Waterloo, 1993.

[105] A. J. McAuley. Dynamic asynchornous logic for high-speed CMOS systems. *IEEE Journal of Solid-State Circuits*, 27(3):382–288, March 1992.

[106] M. C. McFarland. Formal verification of sequential hardware: A tutorial. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 12(5):633–654, May 1993.

[107] E. Meijer. Hazard algebra for asynchronous circuits. In *1992 Glassgow Workshop on Functional Programming, Springer Workshop*, 1992.

[108] T. F. Melham. Abstraction mechanism for hardware verification. In G. Birtwistle and P. A. Subrahmanyam, editors, *VLSI Specification, Verification, and Synthesis*, pages 129–157. Kluwer Academic Publishers, 1988.

[109] T. F. Melham. *Higher Order Logic and Hardware Verification.* Cambridge University Press, 1993.

[110] T. H.-Y. Meng, R. W. Brodersen, and D. G. Messerchmitt. Automatic synthesis of asynchronous circuits from high-level specification. *IEEE Transaction on Computer-Aided Design*, 8(11):1185–1205, November 1989.

[111] R. E. Miller. *Switching Theory, Volume II: Sequential Circuits and Machines.* Willey, 1965.

[112] G. J. Milne. CIRCAL: A calculus for circuit description. *INTEGRATION, VLSI Journal*, 1(3):121–160, July 1983.

[113] G. J. Milne. Towards verifiably correct VLSI design. In G. J. Milne and P. A. Subrahmanyam, editors, *Formal Aspects of VLSI Design*, pages 1–22. Elsevier Science Publishers B. V., 1986.

[114] G. J. Milne. Design for verifiability. In M. Leeser and G. Brown, editors, *Hardware Specification, Verification and Synthesis: Mathematical Aspects*, pages 1–13. Springer-Verlag, 1990.

[115] R. Milner. *A Calculus of Communicating Systems.* Springer-Verlag, 1980. Lecture Notes in Computer Science vol. 92.

[116] B. Mishra and E. M. Clarke. Hierarchical verification of asynchronous circuits using temporal logic. *Theor. Comput. Sci.*, 38:269–291, 1985.

[117] C. E. Molnar, T.-P. Fang, and F. U. Rosenberger. Synthesis of delay-insensitive modules. In H. Fuchs, editor, *1985 Chappel Hill Conference on Very Large Scale Integration*, pages 67–86. Computer Science Press., 1985.

[118] C. E. Molnar and H. Schols. The design problem SCPP-A. Unpublished monograph, April 1995.

[119] C. W. Moon, P. R. Stephan, and R. K. Brayton. Synthesis of hazard-free asynchronous circuits from graphical specifications. In *Proceedings of the International Conference on Computer-Aided Design*, pages 322–325, 1991.

[120] J. D. Morison and A. S. Clarke. *ELLA 2000: A Language for Electronic System Design.* McGraw-Hill, 1994.

[121] D. E. Muller and W. S. Bartkey. A theory of asynchronous circuits. In *Proceedings of International Symposium of the Theory of Switching*, pages 204–243, 1959.

[122] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4), April 1989.

[123] S. Naher and C. Uhrig. *The LEDA User Manual: Version R 3.3*, 1991.

[124] C. D. Nielsen, J. Staunstrup, and S. R. Jones. Potential performance advantage of delay insensitivity. In *IFIP Workshop on Silicon Architectures for Neural Nets*, St. Paul-de-Vence, France, Nov. 1990. ftp:// ftp.id.dth.dk/ pub/Async/neunet.ps.

[125] S. M. Nowick and B. Coates. Automated design of high-performance unclocked state machines. In *TAU93*, 1993.

[126] S. M. Nowick and D. L. Dill. Synthesis of asynchronous state machines using a local clock. In *Proceedings of the 1991 International Conference on Computer Design: VLSI in Computers and Processors*, pages 192–197, 1991.

[127] A. Olsen, O. Færgemand, B. Møller-Pedersen, R. Reed, and J. R. W. Smith. *Systems Engineering using SDL-92.* Elsevier Science B.V., 1994.

[128] S. M. Ornstein, M. J. Stucki, and W. A. Clark. A functional description of macromodules. In *AFIPS Conference Proceedings 1967 Spring Joint Computer Conference*, pages 337–355. Academic Press, April 1967.

[129] L. C. Paulson. Introduction to Isabelle. Technical Report 280, Computer Laboratory University of Cambridge, 1994.

[130] N. C. Paver, P. Day, S. B. Furber, and J. V. Woods. Register locking in an asynchronous microprocessor. In *Proceedings of ICCD'92*, October 1992.

[131] J. F. Peters and B. Rahardjo. Higher order logic in reasoning about asynchronous circuits. In *IEEE WESCANEX'95 Proceedings*, pages 74–78, 1995.

[132] J. L. Peterson. *Petri Net Theory and the Modelling of Systems.* Prentice-Hall, Inc., 1981.

[133] D. Pountain. A call to ARM. *Byte*, pages 293–298, November 1992.

[134] D. Pountain. Computing without clocks. *Byte*, pages 145–150, January 1993.

[135] W. Prasetya. UNITY in HOL. Personal communication, 1994.

[136] O. Pulkkinen and K. Kronlof. Integration of SDL and VHDL for high-level digital design. In *Proceedings of the European Design Automation Conference*, pages 624–629, 1992.

[137] O. Pulkkinen and K. Kronlof. *SDL-VHDL Integration.* John Wiley & Son, 1992.

[138] B. Rahardjo. Asynchronous hardware design: An overview. Technical Report UMECE TR-92-215, Electrical and Computer Engineering, University of Manitoba, 1993.

[139] B. Rahardjo. Hardware equivalences. Technical Report UMECE TR-95-002, Electrical and Computer Engineering, University of Manitoba, 1995.

[140] B. Rahardjo. SPIN as a hardware design tool. In *First SPIN Workshop, Montreal*, October 1995.

[141] B. Rahardjo and R. D. McLeod. Verification of speed-independent asynchronous circuits with protocol validation tools. In *Proceedings of IEEE Pacific Rim Conference on Communications, Computers, and Signal Processing*, pages 257–260, 1995.

[142] B. Rahardjo and R. D. McLeod. Design and analysis of a counterflow pipeline processor in SDL. *Journal of Microelectronic and Systems Integration*, 4(1):33–41, 1996.

[143] B. Rahardjo, J. F. Peters, and R. D. McLeod. Communicating processes in designing asynchronous circuits. *IEEE Aerospace and Electronic Systems Magazine*, pages 8–11, 1995.

[144] M. Rem, J. L. A. van de Snepscheut, and J. T. Udding. Trace theory and the definition of hierarchical components. In R. Bryant, editor, *Third Caltech Conference on VLSI.* Computer Science Press, Inc., 1983.

[145] M. Roncken and R. Saejis. Linear test times for delay-insensitive circuits: A compilation strategy. In *IFIP Working Conference on Asynchronous Design Methodologies*, March 1993.

[146] J. Rushby, F. von Henke, and S. Owre. An introduction to formal specification and verification using EDHM. Technical Report SRI-CSL-91-2, Computer Science Laboratory, SRI International, February 1991.

[147] K. Schneider, R. Kumar, and T. Kropf. Modelling generic hardware structures by abstract datatypes. In *International Workshop on Higher Order Logic Theorem Proving and Its Applications*, pages 419–429. Elsevier Science Publishers, September 1992.

[148] K. Schneider, R. Kumar, and T. Kropf. Why hardware verification needs more than model checking. e-mail: schneide@ira.uka.de, 1994.

[149] C.-J. Seger. An introduction to formal hardware verification. Technical Report TR-92-13, Departement of Computer Science, University of British Columbia, 1992.

[150] C.-J. H. Seger. Voss - a formal hardware verification system user's guide. Technical Report TR93-45, Dept. of Computer Science, University of British Columbia, December 1993.

[151] C.-J. H. Seger. Voss - a formal verification system user's guide. Technical Report 93-45, Dept. of Computer Science, University of British Columbia, December 1993.

[152] C. L. Seitz. Self-timed VLSI systems. In *Proceeding of the Caltech Conference on Very Large Scale Integration*, pages 345–355, January 1979.

[153] C. L. Seitz. System timing. In C. Mead and L. Conway, editors, *Introduction to VLSI Systems*, chapter 7, pages 218–262. Addison-Wesley, 1980.

[154] R. F. Sproull, I. E. Sutherland, and C. E. Molnar. The counterflow pipeline processor architecture. *IEEE Design & Test of Computers*, Fall:48–50, 1994.

[155] J. Staunstrup. *A Formal Approach to Hardware Design.* Kluwer Academic Publishers, 1994.

[156] V. Stavridou. *Formal Methods in Circuit Design.* Cambridge University Press, 1993.

[157] V. Stavridou. Formal methods and VLSI engineering practice. *The Computer Journal*, 37(2):96–113, 1994.

[158] K. S. Stevens. *Practical Verification and Synthesis of Low Latency Asynchronous Systems.* PhD thesis, University of Calgary, September 1994.

[159] M. J. Stucki, S. M. Ornsteib, and W. A. Clark. Logical design of macromodules. In *AFIPS Conference Proceedings 1967 Spring Joint Computer Conference*, pages 357–363. Academic Press, April 1967.

[160] I. Sutherland. Micropipeline. *CACM*, 32(6):720–738, 1989.

[161] I. Sutherland and C. A. Mead. Microelectronics and computer science. *Scientific American*, pages 210–228, September 1977.

[162] Telelogic. *SDT 3.0 User Guide*, 1995.

[163] D. E. Thomas and P. Moorby. *The Verilog Hardware Description Language.* Kluwer Academic Publishers, 1991.

[164] J. A. Tierno, A. J. Martin, and D. Borkovic. An asynchronous microprocessor in gallium arsenide. Technical Report CS-TR-93-38, Dept. of Computer Science, California Institute of Technology, November 1993.

[165] C. H. van Berkel, R. Burgess, J. Kessels, A. Peeters, M. Roncken, and F. Saejis. A fully asynchronous low-power error corrector for digital compact cassette. In *IEEE International Solid-State Circuits Conference*, 1994.

[166] C. H. van Berkel and R. W. J. J. Saeijs. Compilation of communicating process into delay-insensitive circuits. In *Proceedings of the International Conference on Computer Design*, pages 157–162. Computer Society Press, 1988.

[167] K. van Berkel. Handshake circuits: An asynchronous architecture for VLSI programming. In *Proceedings of the VII Banff Workshop*, 1993.

[168] K. van Berkel. VLSI programming of a modulo-N counter with constant response time and constant power. In *IFIP Working Conference on Asynchronous Design Methodologies*, March 1993.

[169] J. L. A. van de Snepscheut. Deriving circuits from programs. In *Third CalTech Conference on Very Large Scale Integration*, pages 241–256. Computer Science Press, Inc., 1983.

[170] J. L. A. van de Snepscheut. *Trace Theory and VLSI Design.* Lecture Notes in Computer Science vol. 200. Springer Verlag, 1985.

[171] J. P. Van Tassel. *Femto-VHDL: The Semantics of a Subset of VHDL and its Embedding into HOL.* PhD thesis, Gonville and Caius College, University of Cambridge, July 1993.

[172] V. I. Varshavsky, editor. *Self-timed Control of Concurrent Processes.* Kluwer Academic Publisher, 1990.

[173] L. Wall and R. L. Schwartz. *Programming Perl.* O'Reilly & Associates Inc., 1991.

[174] N. Weste and K. Eshraghian. *Principles of CMOS VLSI Design: A Systems Perspective.* Addison-Wesley, 1985.

[175] A. Wolfe. Intel fixes a pentium FPU glitch. *EE Times*, page 1, Nov. 1994. issue 822.

[176] A. Yakovlev. Designing control logic for counterflow pipeline processor using petri nets. Unpublish monograph, Department of Computer Science, University of Newcastle upon Tyne, England, May 1995.

Appendix A

A Hazard-free C-element

A hazard-free gate-level implementation of a C-element is illustrated below. This implementation was suggested by Oleg Mayevsky.

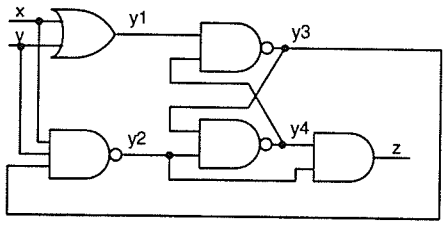


Figure A.1: Hazard-free C-element

$$
\begin{aligned}
y_1 &= x + y \\
y_2 &= not(x \cdot y \cdot y_3) \\
y_3 &= not(y_1 \cdot y_4) \\
y_4 &= not(y_2 \cdot y_3) \\
z &= y_2 \cdot y_4
\end{aligned}
\tag{A.1}
$$

A PROMELA model of the circuit is listed below.

```
bit x, y, y1,y2,y3,y4, z, oldz;
bit wait;

#define AND2(x,y,out)     (out != (x&&y)) -> out = x&&y
#define NAND2(x,y,out)    (out == (x&&y)) -> out = 1-(x&&y)
#define NAND3(x,y,z,out) (out == (x&&y&&z)) -> out = 1-(x&&y&&z)
#define OR2(in1,in2,out) (out != (in1||in2)) -> out = in1||in2
#define OR3(in1,in2,in3,out) (out != (in1||in2||in3)) -> out = in1||in2||in3
#define INV(in,out)      (out == in) -> out = 1-in

proctype stimulus()
{
   do
   ::   wait == 0 ->
        oldz=z;
        if
        :: x = 1-x
        :: y = 1-y
        fi;
        wait = 1
   od
}

proctype monitor()
{
   do
   :: wait == 1 ->
      if
      :: (x==0 && y==0 && z==0)
      :: (x==1 && y==1 && z==1)
      :: (x!=y && z==oldz)
      fi;
      wait = 0
   od
}

init
{
        atomic{y1=0; y2=1; y3=1; y4=0};
        run stimulus();
        run monitor();
```

```
        do
        :: OR2(x,y,y1)
        :: NAND3(x,y,y3,y2)
        :: NAND2(y1,y4,y3)
        :: NAND2(y2,y3,y4)
        :: AND2(y2,y4,z)
        od
}

never {
        do
        :: skip
        :: (x != y) -> goto accept0
        :: (x == y) -> goto accept1
        od;
accept0:
        do
        :: (z != oldz)
        od;
accept1:
        do
        :: (z != x || z != y)
        od
}
```

Appendix B

Another Hazardous C-element

Another hazardous gate-level implementation of a C-element is illustrated below.

Another hazardous C-element implementation is shown below.

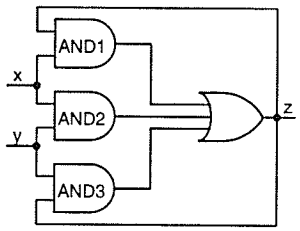


Figure B.1: Hazardous C-element

$$z' = x \cdot y + x \cdot z + y \cdot z \tag{B.1}$$

The PROMELA model is listed below.

```
bit x, y, z1,z2,z3, z, oldz;
bit wait;

#define AND(x,y,out)     (out != (x&&y)) -> out = x&&y
#define OR3(in1,in2,in3,out) (out != (in1||in2||in3)) -> out = in1||in2||in3
#define INV(in,out)      (out == in) -> out = 1-in

proctype stimulus()
```

```
{
   do
   ::   wait == 0 ->
        oldz=z;
        if
        :: x = 1-x
        :: y = 1-y
        fi;
        wait = 1
   od
}

proctype monitor()
{
   do
   :: wait == 1 ->
      if
      :: (x==0 && y==0 && z==0)
      :: (x==1 && y==1 && z==1)
      :: (x!=y && z==oldz)
      fi;
      wait = 0
   od
}

init
{
        run stimulus();
        run monitor();
        do
        :: AND(x,z,z1)
        :: AND(x,y,z2)
        :: AND(y,z,z3)
        :: OR3(z1,z2,z3,z)
        od
}

never {
        do
        :: skip
        :: (x != y) -> goto accept0
        :: (x == y) -> goto accept1
```

```
        od;
accept0:
        do
        :: (z != oldz)
        od;
accept1:
        do
        :: (z != x || z != y)
        od
}
```

Appendix C

A simple state space search program

A simple full state space exploration algorithm is described in [73]. The following is a slightly modified pseudo-code of the algorithm.

```
state_search()
{
   initialize a working queue W;
   initialize set A for previously analyzed state;
   analyze();
}

/* exhaustive state-space search */
analyze()
{
   if (set W is empty) return; /* done */
   pick an element q from W; /* FIFO or LIFO */
   add q to set A;
   if (q is an error_state)
      report_error();
   else
   {
      for each successor state s of q
         if (s is not in A or W)
         {
            add s to W;
```

```
            analyze();
        }
    }
    delete q from W;
}
```

Based on the above algorithm, a simple state space search program has been constructed. The program is implemented in C++ and uses the LEDA library [123], which results in a source code that has a structure close to its pseudo-code.

A two-input AND gate is used as an example of system under study. The AND gate has four states, which are shown in Figure C.1. In this figure, we can see that from state "0" we can move to state "1" or "2". That is, state "1" and "2" are the successors of state "0". Similarly, from state "1" we can move to state "0" or "3", and so on.

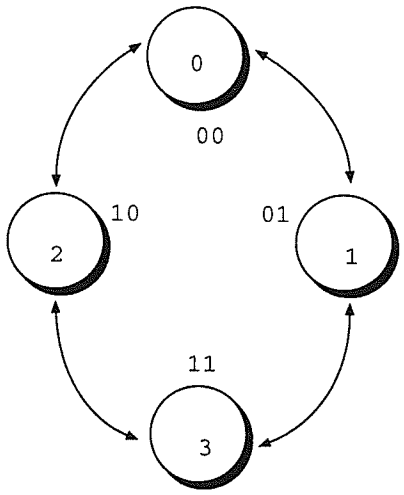


Figure C.1: States of two-input AND gate

C.1 Program Listing

The following is a listing of the C++ source code of our state space search program. The description of the circuit under study (AND gate) is embedded in the code as

a procedure. In a future version, this routine should be replaced by a more general procedure, which can read the description of the circuit under study in a standard format. Error handling routines are not provided in this sample program.

```
// Simple Reachability Analysis - State Space Exploration
// Budi Rahardjo
// Algorithm is based on
// G. Holzmann, "Design and Validation of Computer Protocols"
//
#include <LEDA/stack.h>
#include <LEDA/set.h>
#include <LEDA/list.h>

void analyze(void);
list<int> successor(int);

stack<int> W;      // Working queue
set<int> Wset;     // Set for states to be visited
set<int> A;        // Set of visited states

void analyze()
{
   int q;
   int x;
   list<int> S;

   cout << "Analyze()" << endl;
   if (W.empty()) return;
   q = W.pop(); Wset.del(q);   // pull q, an element from working queue
                               // Stack is used -> DFS
   cout << "Visiting state " << q << endl;
   A.insert(q);                // Insert q in the set of visited states
   S = successor(q);           // generate a list of successors of q
   forall(x,S) {               // for each successor of q
      if ( (! A.member(x)) && (! Wset.member(x)) ) {
         cout << "   Pushing state " << x << endl;
         W.push(x); Wset.insert(x); // state x has not been visited
                               // add it to the working queue
         analyze();
      } else {
         cout << "   State "<< x << " is already visited. Skipped." << endl;
      }
   }
}
```

```
list<int> successor(int q)
{
   // example of system under test
   // in this particular case, it's a 2-input AND gate
   // which has four states
   // States are hand coded: 00 = 0, 01 = 1, 10 = 2, 11 = 3
   // successors are also hand coded
   // successor() produces a list of next states

   list<int> SL;

   cout << "   generating successor of state ";
   switch (q) {
      case 0: cout << "0";
         SL.append(1); SL.append(2); break;
      case 1: cout << "1";
         SL.append(0); SL.append(3); break;
      case 2: cout << "2";
         SL.append(0); SL.append(3); break;
      case 3: cout << "3";
         SL.append(1); SL.append(2); break;
      default: cout << "*WARNING* default";
         SL.append(0);
   };
   cout << endl;
   return SL;
}


main()
{
   int x;

   W.push(0); Wset.insert(0); // start with initial state (0)
   analyze();

   cout << endl << "State space search is done." << endl;
   // print visited states
   cout << "Visited states: ";
   forall(x,A) { cout << x << " "; };
   cout << endl;
}
```

C.2 Sample run

The source code was complied with a C++ compiler. An executable program was produced and executed. The following is a sample output of the program.

```
Analyze()
Visiting state 0
   generating successor of state 0
   Pushing state 1
Analyze()
Visiting state 1
   generating successor of state 1
   State 0 is already visited. Skipped.
   Pushing state 3
Analyze()
Visiting state 3
   generating successor of state 3
   State 1 is already visited. Skipped.
   Pushing state 2
Analyze()
Visiting state 2
   generating successor of state 2
   State 0 is already visited. Skipped.
   State 3 is already visited. Skipped.
   State 2 is already visited. Skipped.

State space search is done.
Visited states: 0 1 2 3
```

C.3 Short Discussions

The use of the LEDA library, which provides various data structures, simplifies the writing of the program in which the source code is very close to its algorithm.

The program performs a *Depth First Search* (DFS) search. This is shown by the sample run, in which the order of visited states is 0, 1, 3, and 2. The choice of data structure for W (working queue) determines the search algorithm. In this particular

example, W was implemented with a *stack*. To perform *Breadth First Search* (BFS), W should be implemented with a *buffer*.

The system under study, a two-input AND gate, is very small. The reason for selecting the AND gate is for clarity. The states and their successors of the AND gate were hand coded in the program. In the future, a more general procedure should be used instead.

There are other issues that have to be considered if this approach is going to be used in real designs, which usually have large state spaces. These issues include the use of efficient data structures, the use of various state reduction techniques, and the use of techniques to perform *on-the-fly* state construction.

Appendix D

Script: Generate signals

The following is a perl-script[1] used to generate permutation of parameters used in "*procData*" signal for validating block type `Ptype` discussed in section 5.4. The script generates 98304 combinations; $3 * 2^{15}$.

```
#! /opt/gnu/bin/perl

@INSTRUCTIONS = ('NOP', 'LDD', 'XXX');
@REGISTERS = ('A', 'B');
@VALUES = (55,1);
@BOOLEAN = ('true', 'false');

foreach $inst (@INSTRUCTIONS) {
   foreach $reg1 (@REGISTERS) {
      foreach $val1 (@VALUES) {
         foreach $valid1 (@BOOLEAN) {
   foreach $reg2 (@REGISTERS) {
      foreach $val2 (@VALUES) {
         foreach $valid2 (@BOOLEAN) {
   foreach $reg3 (@REGISTERS) {
      foreach $val3 (@VALUES) {
         foreach $valid3 (@BOOLEAN) {

   foreach $res1 (@REGISTERS) {
      foreach $resval1 (@VALUES) {
```

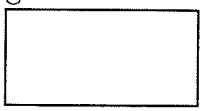
[1]Perl is a scripting language developed by Larry Wall [173].

```
         foreach $resvalid1 (@BOOLEAN) {
   foreach $res2 (@REGISTERS) {
      foreach $resval2 (@VALUES) {
         foreach $resvalid2 (@BOOLEAN) {

print 'define-signal procData((. ';
print "'$inst', '$reg1', $val1, $valid1, '$reg2', $val2, $valid2, ";
print "'$reg3', $val3, $valid3 .),(. ";
print "'$res1', $resval1, $valid1, ";
print "'$res2', $resval2, $valid2, ";
print ".))\n";

         } # $resvalid2
      } # $resval2
   } # $res2
         } # $resvalid1
      } # $resval1
   } # $res1

         } # $valid3
      } # $val3
   } # $re3g
         } # $valid2
      } # $val2
   } # $reg2
            } # $valid1
         } $val1
      } # $reg1
   } # $inst
```

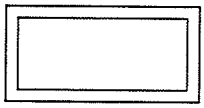
Appendix E

Selected SDL Symbols

The following is a list of selected SDL symbols which are used in this thesis.

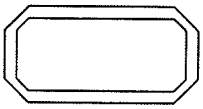


block reference

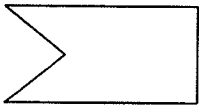


block type reference

process reference



process type reference

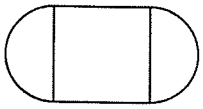


input

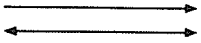
output

procedure call

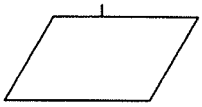
procedure reference



procedure start



non-delaying channel

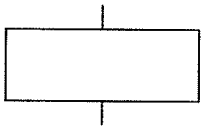


save

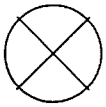
start

stop

state



task



return

text

Appendix F

The Syntax for Real-Time CSP

The syntax of Real-Time CSP in BNF form is listed below.

P ::=	
STOP	{never engages in any action}
| $\perp$	{empty behavior}
| SKIP	{terminate successfully}
| wait t	{delay t ticks of clock}
| a $\rightarrow$ P	{event a then process P}
| P ; P	{P before P}
| wait t ; P	{delay t before P}
| P $\Box$ P	{deterministic choice}
| P $\sqcap$ P	{non-deterministic choice}
| P || P	{P in parallel with P}
| P \ A	{hiding int. action A}
| *P	{repeat P}
| b*P	{b then *P}
| if cond then P else P	
a ::= start P	{instant when P starts}
| end P	{instant when P ends}
| ch ? msg	{on channel ch input msg}
| ch ! msg	{on channel ch output msg}
cond := success P	{P satisfies its constraints}
| ontime P	{success P $\wedge$ start P $<$ end P $\wedge$ end P $<$ (start P + limit P)}
| other	{boolean condition}

Appendix G

Proving Didel and TwoInv

In Chapter 6 we discussed that a circuit (called "*TwoInv*") which consists of two inverters in series is an implementation of "*Didel*" element. Suppose the output of the inverter is delayed by one unit-time, then we can prove that the "*TwoInv*" is a "*Didel*" element with its output delayed by two unit-time delays. The proof (following [108]) is sketched below.

Given

$$\vdash \text{Inv}(i, o) = \forall t.\ o(t+1) = \neg i\ t \tag{G.1}$$

$$\vdash \text{TwoInv}(i, x, o) = \text{Inv}(i, x) \wedge \text{Inv}(x, o) \tag{G.2}$$

$$\vdash \text{Didel}(i, o) = \forall t.\ o(t+t') = i\ t \tag{G.3}$$

we want to prove that

$$\forall i\ o.\ \text{Didel}(i, o) \equiv \exists x.\ \text{TwoInv}(i, x, o) \tag{G.4}$$

The proof proceeds as follows:

1. By definition of "Inv"

   $\exists x.\ \text{TwoInv}(i, x, o) \equiv \exists x.\ (\forall t.\ x(t+1) = i(t) \wedge \forall t.\ o(t+1) = x(t))$

2. By symmetry of equality

   $\exists x.\ \text{TwoInv}(i, x, o) \equiv \exists x.\ (\forall t.\ x(t+1) = i(t) \wedge \forall t.\ x(t) = o(t+1)$

3. By rewriting with the equation '$\forall t.\ x(t) = o(t+1)$' yields

   $\exists x.\ \text{TwoInv}(i, x, o) \equiv \exists x.\ (\forall t.\ o((t+1)+1) = i(t) \wedge \forall t.\ x(t) = o(t+1))$

4. Since x does not occur in the left hand conjunct, the scope of existensial quantifier can be limitted to the right hand conjunct.

   $\exists x.\ \text{TwoInv}(i, x, o) \equiv \forall t.\ o((t+1)+1) = i(t) \wedge \exists x.\ (\forall t.\ x(t) = o(t+1))$

5. '$\exists x.\ (\forall t.\ x(t) = o(t+1))$' is a tautology, the equivalence reduces to

   $\exists x.\ \text{TwoInv}(i, x, o) \equiv \forall t.\ o((t+1)+1) = i(t)$

6. Simplifying '$((t+1)+1)$' yields

   $\exists x.\ \text{TwoInv}(i, x, o) \equiv \forall t.\ o(t+2) = i(t)$

7. Taking $t' = 2$, the equivalence becomes

   $\exists x.\ \text{TwoInv}(i, x, o) \equiv \text{Didel}(i, o)$

Thus, "*TwoInv*" is a "*Didel*" element with $t' = 2$ (i.e. output is delayed by 2 units).

Glossary

ADT ADT stands for Abstract Data Types, a mechanism to define new data types in SDL.

BDD BDD stands for Binary Decision Diagram, a compact representation of Boolean functions in a tree-like form.

CAD CAD stands for Computer-Aided Design. CAD refers to the design of hardware with the aid of computers.

CASE CASE stands for Computer-Aided Software Engineering.

CCITT CCITT stands for Comité Consultatif International Télégraphique et Téléphonique. Now, it is called ITU-T.

CCS CCS stands for Calculus of Communicating Systems.

CFPP CFPP stands for Counterflow Pipeline Processor.

CSP CSP stands for Communicating Sequential Processes.

EFSM EFSM stands for Extended Finite State Machine. An EFSM is a state machine augmented with internal data storage (e.g. SDL internal variables) and a communication mechanism.

FFT FFT stands for Fast Fourier Transform.

FSM FSM stands for Finite State Machine.

HDL HDL stands for Hardware Description Language.

HOL HOL stands for Higher-Order Logic. HOL is also the name of a theorem proving environment.

ITU International Telecommunication Union.

ITU-T ITU Telecommunication Standardization Sector. Formerly it is called CCITT.

MSC MSC stands for Message Sequence Chart, a mean for the description, graphical visualization, of selected traces within distributed systems.

RTCSP RTCSP stands for Real-Time Communicating Sequential Processes. RTCSP extends Hoare CSP by providing time-critical construct for modeling timeouts, delays, and parallel processing.

SCPP SCPP stands for Sproull's Counterflow Pipeline Processor.

SDL SDL stands for Specification and Description Language. SDL is standardized by the ITU-T as Recommendation Z.100. It is suited for event-driven systems which communicate via message passing. SDL is commonly used in telecommunication applications.

SDL-GR SDL-GR stands for SDL - Graphical Representation, a graphical representation of SDL.

SDL-PR SDL-PR stands for SDL - Phrase Representation, a textual representation of SDL.

SDT SDT stands for SDL Design Tool, a set of tools developed by Telelogic.

STG STG stands for Signal Transition Graph. STG is a derivation of Petri Nets.

VHDL VHDL stands for VHSIC Hardware Description Language. VHDL is a popular language to model hardware (circuits).

VHSIC VHSIC stands for Very High Speed Integrated Circuit.

VLSI VLSI stands for Very Large Scale Integration.