FORMAL VERIFICATION AND TESTING OF ASYNCHRONOUS CIRCUITS

Oriol Roig i Mansilla
Universitat Politècnica de Catalunya
Barcelona. May, 1997

UPC

A THESIS SUBMITTED IN FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE
Doctor en Informàtica

PREFACE

The advances in *Very Large Scale Integration* (VLSI) technology in the last years has led to faster, more complex and less consuming digital circuits. Another key point in the revolutionary changes produced in electronics has been the drastic cost reduction of manufactured products. The cost of a chip is mainly concentrated in the design, the layout mask (i.e. the "die") fabrication and testing. Surprisingly, manufacturing the chip itself is much cheaper since a VLSI circuit is, roughly speaking, a piece of sand.

Most of nowadays digital circuits are synchronous because a variety of reasons:

- Synchronous circuits can be fabricated in relatively short time thanks to numerous available *Computer-Aided Design* (CAD) tools. Many of these tools are editors, since typically earlier design stages are still held manually.
- Circuit "correctness" is usually assumed by using presumably correct synthesis tools plus extensive hand-driven and/or pseudo-random simulation. This method, highly dependent on designer's skills, can be easily used but it does not constitute a formal prove of correctness. As a sample, one of the latest and most famous undetected bugs, uncovered by extensive simulation, has been the faulty divider of earlier Pentium microprocessors. The use of formal verification methods is currently quite reduced in the industry.
- There are approaches that highly enhance the testability of synchronous circuits. It has even been defined a standard, the P1149, to describe how such techniques as well as the communication with test equipment should be done.

The advantages of synchronous circuits are consequence, in a measure, of very restrictive design rules that simplify, among other issues, timing analysis. In synchronous circuits all feedbacks are cut by flip-flops, which allows stepping computation in cycles. Since the only timing concern is that by the end of the cycle the circuit must be stable, synchronous circuit design and analysis is straightforward if compared with asynchronous design.

The main reason that eased synchronous system design is turning out to be one of its capital drawbacks as VLSI technology develops, and the situation will probably be worse with the advent of *Ultra Large Scale Integration* (ULSI) [Boh96]. The fact that a signal cannot be exactly distributed at the same time at every point of a chip cannot be longer neglected as cycle time shortens. A possible solution to this problem might be local rather than global synchronization.

Asynchronous systems use local synchronization to interact. Computation starts as soon as there is new available data and communication is done by local handshake signals. This modus operandi makes asynchronous circuits highly concurrent systems where everything is done, at least in theory, *as soon as possible*. Absence of clock skew, high degree of modularity, potential lower power consumption and electro-magnetic radiation, or average-case performance are, among others, advantages of asynchronous circuits. Consequently, asynchronous circuits potentially provide means of overcoming the arising disadvantages of synchronous systems and taking maximum benefit from leading-edge VLSI and ULSI technology.

Concurrent systems are difficult to design and to (formally) prove to be correct. Moving myriad designers to a challenging asynchronous style is an activity that will not succeed unless the designer's task is eased at maximum, as it is the case of nowadays synchronous design. Providing methodologies and tools that do the dirty work is, therefore, even more necessary than it has been in clocked systems.

The work to be introduced proposes novel techniques for the automatic verification an test pattern generation of asynchronous circuits. From the practical point of view, the presented approaches are not just a scientific divertimento, but they have resulted in the implementation of two new CAD tools.

CONTENTS

PREFACE iii

ACKNOWLEDGEMENTS ix

1 INTRODUCTION 1
- 1.1 Integrated circuit design methodology 1
  - 1.1.1 Synthesis 1
  - 1.1.2 Verification 3
  - 1.1.3 Testing 3
- 1.2 Why asynchronous? 5
- 1.3 Asynchronous circuit modeling 7
- 1.4 Overview 9
  - 1.4.1 Specification validation 10
  - 1.4.2 Asynchronous circuit verification 10
  - 1.4.3 Testing of asynchronous circuits 10
- 1.5 Structure of the work 11

2 MATHEMATICAL BACKGROUND 13
- 2.1 Preliminaries 13
- 2.2 Sets, relations and functions 14
- 2.3 Logic functions 16
- 2.4 Boolean algebra 17
  - 2.4.1 Switching algebra 18
  - 2.4.2 Algebra of classes (subsets of a set) 18
  - 2.4.3 Algebra of logic functions 18
  - 2.4.4 Stone's representation theorem 18
- 2.5 Ternary algebra 19
- 2.6 Making it efficiently: BDDs 19
  - 2.6.1 Reduced and ordered BDDs 20
  - 2.6.2 Boole's expansion theorem 21
  - 2.6.3 Boolean operations with ROBDDs 21

3 SPECIFICATION VALIDATION 23
- 3.1 Introduction 23
  - 3.1.1 Previous work 24
  - 3.1.2 Overview 25
- 3.2 Petri nets 25

3.2.1 Modeling safe Petri nets 26
3.2.2 Firing Petri net transitions 27
3.3 Signal Transition Graphs (STG) 28
3.3.1 Modeling safe STGs 29
3.3.2 Firing STG transitions 30
3.4 State graphs 30
3.4.1 Projection of a state graph 30
3.4.2 State graph associated to a Petri net or an STG 31
3.5 Net traversal and reachable markings 31
3.6 Petri net reductions 33
3.7 Improving Petri net traversal 34
3.7.1 Chaining 34
3.7.2 Toggling 36
3.7.3 Improved BFS algorithm 36
3.8 Verification of properties 38
3.8.1 Safeness 39
3.8.2 Liveness 39
3.8.3 Persistence 40
3.8.4 Consistence 42
3.8.5 Complete State Coding 43
3.9 Results 43
3.9.1 Petri nets 43
3.9.2 Signal Transition Graphs 45
3.10 Conclusions 46

4 ASYNCHRONOUS CIRCUIT VERIFICATION 47
4.1 Introduction 47
4.1.1 Previous work 48
4.1.2 Overview 49
4.2 Notation and terminology 49
4.3 Speed-independent circuit modeling 50
4.3.1 Gate modeling 51
4.4 Environment and circuit composition 53
4.5 Verification of speed-independence 54
4.5.1 Failure states 55
4.5.2 Semi-modularity 56
4.6 Flat verification of speed-independent circuits 57
4.6.1 Error diagnosis 57
4.7 Hierarchical verification of speed-independent circuits 59
4.7.1 Functional and behavioral correctness 60
4.7.2 Reduction to complex gates 63
4.7.3 Projections adding nondeterminism 64
4.7.4 Why is hierarchical verification exact? 65
4.7.5 Verification of isomorphic subcircuits 69
4.8 Verification of properties 70
4.8.1 Deadlock freeness 70

4.8.2 Home state 70
4.8.3 Verification of properties on the collapsed circuit 71
4.9 Results 72
4.10 Conclusions 75

5 TEST PATTERN GENERATION FOR ASYNCHRONOUS CIRCUITS 77
5.1 Introduction 77
5.1.1 Terminology 77
5.1.2 Previous work 78
5.1.3 Overview 79
5.2 Faults and fault models 80
5.3 Asynchronous circuit-under-test behavior 82
5.3.1 Circuit model 84
5.3.2 Circuit State Graph (*CSG*) 84
5.3.3 *CSG* in test mode (*TCSG*) 85
5.4 Synchronous abstraction of the *TCSG* 86
5.4.1 Estimation of the test cycle 86
5.4.2 Practical computation of the *CSSG* 86
5.5 Automatic test pattern generation 88
5.5.1 Fault activation 88
5.5.2 State justification 88
5.5.3 State differentiation 89
5.6 Improving ATPG performance 89
5.6.1 Ternary simulation 90
5.6.2 Fault simulation 91
5.6.3 Random TPG 92
5.7 Results 93
5.7.1 Discussion 95
5.8 Conclusions 95

6 CONCLUSIONS AND FUTURE WORK 97
6.1 Main contributions 97
6.2 Open problems and future work 98
6.2.1 Verification 98
6.2.2 Testing 98

REFERENCES 99

LIST OF FIGURES 107

LIST OF TABLES 109

INDEX 111

ACKNOWLEDGEMENTS

This work would have been infinitely more difficult without the help of many wonderful people. I will try to thank them in the tongue I usually address to them.

En primer lloc, al principal culpable de que aquest treball arribi mai a ser una tesi, en Jordi Cortadella. L'entusiasme i esforç que ha posat en la seva feina com a director mereixen molt més que aquestes quatre ratlles d'agraïment. M'ha ensenyat a fer recerca, m'ha corregit, m'ha adressat cap al bon camí, ens hem estavellat plegats contra algun escull i, finalment, hem arribat a port (però d'aquí a no res tornem a navegar). Per a ell és la meva més sincera gratitud.

També vull agrair les enriquidores xerrades i els cafès amb n'Enric Pastor i en Marco Antonio Peña. I a tots els del grup de CAD-VLSI, especialment a Rosa Maria Badia, Enric Musoll i Fermín Sanchez, l'haver-me aguantat, entre d'altres coses, el nombrosos assaigs quan s'apropava una presentació. A Abelardo Pardo quiero agradecerle que, a pesar de estar lejos físicamente, siempre ha estado a un simple e-mail de distancia para lo que hiciera falta.

I would also like to thank the valuable comments and discussions on verification and testing with Michael Kishinevsky, Alex Kondratyev, Luciano Lavagno and Alex Yakovlev. I am also indebted to Alex Yakovlev for hosting me in my visit to NewCastle and to Alex Semenov for showing me the "non-universitary" part of the city.

No vull oblidar als companys del Departament d'Arquitectura de Computadors, en especial a la penya del Laboratori de Càlcul, per fer backups i ajudar-me a recuperar el darrer capítol d'aquest treball, que jo inconscientment havia destruït.

This work has been supported in part by the *Ministerio de Educación y Ciencia* of Spain (CICYT), under contracts TIC 91-1036, TIC 94-0531-E, TIC 95-0419 and Acción Integrada HB-1995-0203, and by the *Asynchronous Circuit Design Work Group* under contract Esprit 7225 and Esprit 21949. I am greatly indebted to the *Departament d'Ensenyament de la Generalitat de Catalunya* for financing my doctorate. Various grants from the *Universitat Politècnica de Catalunya* and a 34th Design Automation Conference Travel Grant are also acknowledged.

Finalment vull agrair als meus pares, Felip i Mercè, el suport i la fe cega que han tingut en la meva feina. A mon germà Daniel ser un bandarra collonut, igual que tota la colla de "Pallufoplàstics" que fan que el bàsquet sigui de tot menys seriós. A l'Angelina i l'Apo que em tinguin a casa seva com si fos la meva. Als amics, que ho segueixin essent després de tants anys, en particular als més vells (i bells) Àlvar, Ivan i Enric. I, especialment, a la Maria Jesús que m'aguanti l'humor i les rareses, m'ajudi a tirar endavant i, malgrat tot, m'estimi. Realment hauria estat molt més dur sense tots ells.

I would like to thank all of them for any contribution, and apologize to anyone I could have forgotten. Gràcies. Gracias. Grazie. *Спасибо*.

–"Vós, cavaller, qui haveu rebut l'orde de cavalleria, e sou tengut en opinió de no ésser reprotxe entre los bons cavallers, jo só tramès ací per ambaixador de tota la fraternitat e d'aquell pròsper orde del benaventurat senyor Sant Jordi, que per aquell jurament que fet haveu, que tendreu totes les coses secretes e que per via directa o indirecta, de paraula o per escrit, no ho manifestareu."

E lo cavaller promet per virtut del jurament, complir e servar totes les coses dessús dites, e donen-li los capítols. Aprés que els ha llests, si els accepta, agenolla's en terra davan l'altar o imatge de Sant Jordi, e ab molta honor e reverència rep l'orde de la fraternitat. E si acceptar no la vol, ha tres dies d'espai de pensar-hi, e diu o pot dir: "La mia persona no és disposta per rebre un tan alt orde com és aquest, ple de molta excel·lència e virtut."

Torna a cloure los capítols, escriu dins son nom, e així els torna a trametre per l'ambaixador als de la fraternitat.

Joanot Martorell, "Tirant lo Blanc."

1 INTRODUCTION

"Read them," said the King.

The White Rabbit put on his spectacles. "Where shall I begin, please your Majesty?" he asked.

"Begin at the beginning," the King said gravely, "and go on till you come to the end; then stop."

– Lewis Carrol, "Alice in Wonderland."

1.1 INTEGRATED CIRCUIT DESIGN METHODOLOGY

Very Large Scale Integration (VLSI) circuits are components of most nowadays electronic systems. These circuits contain thousands or millions of electronic devices such as transistors, diodes, capacitors and/or resistors. The design of such circuits is a complicated process that involves many different stages. The high complexity of all these steps requires a significant degree of automation. Much effort has been devised for automating the design process of synchronous circuits, and new commercial tools appear day by day. In the field of asynchronous circuits, though, only recently some automated methodologies have begun to slowly emerge.

Figure 1.1 depicts a typical circuit design task flow [GDKW92], valid both for synchronous and asynchronous circuits. Boxes represent different descriptions of a circuit, from the abstract *initial idea* until the *final product*. Ellipses identify procedures that obtain a more refined description from a previous one, or that check or modify a given description. Each procedure belongs to one of the three major aspects of circuit design: *synthesis*, *verification* and *testing*. Verification assures that synthesis has been correct, whereas testing checks that a physical circuit is a correct translation into silicon of a particular description of a circuit.

1.1.1 Synthesis

Given an initial idea, sometimes rather abstract, of what a circuit should do, human designers must build a specification of that circuit. This specification is a high-level behavioral description of the circuit, and can be written in some high-level language like Verilog [Cad89], VHDL [New88], CSP [Hoa78, Hoa85], etc, or in other high-level formalisms such as Petri nets [Pet62].

High-level synthesis takes that specification and obtains a *data-path* with elements such as registers, multiplexers, adders, multipliers, etc, and a description of the control of those modules.

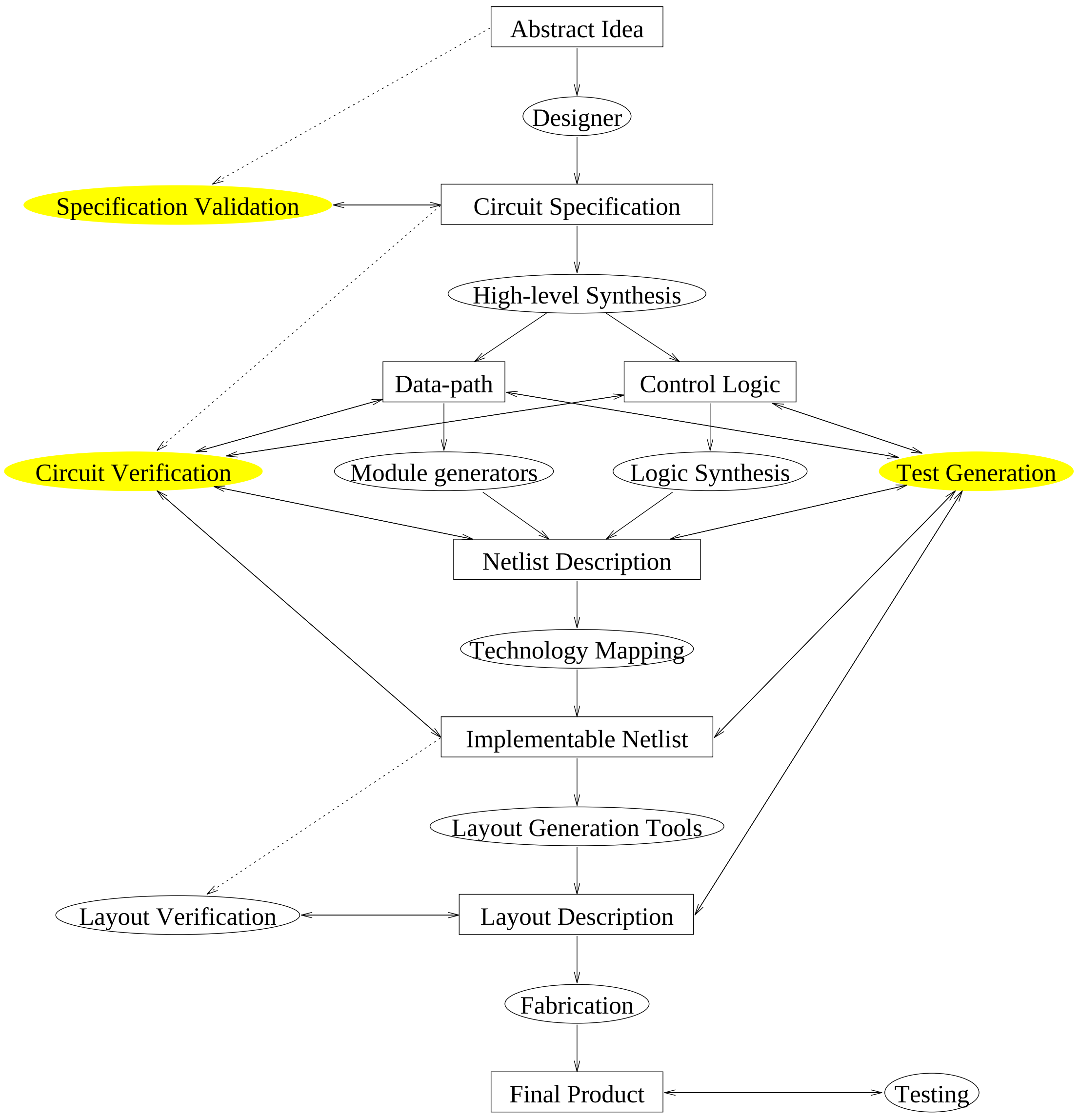


Figure 1.1 Typical circuit design pipeline.

This type of description is often called a *register-transfer level description*. Commonly there are different available modules that perform a same operation, and high-level synthesis algorithms try to combine them in order to generate the smaller, faster and/or less-consuming circuit satisfying the area, throughput and power requirements. An automaton describing how those modules must cooperate to produce the circuit response is also generated.

The goal of *logic-level synthesis* is generating a circuit described as a netlist of gates. The control automaton is translated into a set of equations and these equations are mapped onto an arbitrary set of gates.

Given that set of gates, *technology mapping* will not only try to directly map each arbitrary gate onto a gate in a library of implementable gates, but also split inexistent complex gates into several simpler ones. In addition, groups of simple gates may be combined to form larger complex gates, provided that such gates are available in the implementable library. The data-path modules may also be translated into several gates, but usually they are directly mapped onto a transistor description by the so called *module generators* or by using available module libraries.

The last step consists in obtaining the layout of the circuit. Modules and gates are placed and connected by using placement and routing tools. The result is a mask-level description of the circuit that can be used to fabricate the final product.

1.1.2 Verification

Along the various synthesis steps, an erroneous procedure can contribute to obtain an incorrect circuit description that may invalidate subsequent design stages. Human mistakes in manually performed steps, undetected software bugs in automatic tools or a misuse of those tools are possible sources of error. Therefore, the synthesis process cannot be considered error-free, and it is always desirable to perform some kind of verification after obtaining a new circuit description.

The devil's advocate should state that there is also some chance that the verifier could be wrong, therefore, in theory, no design process can be assured to be error-free. In defense of verification, it must be said that a same design can be verified by different verification tools. As far as their verdict coincided, there is a strong likelihood of both synthesis and verification tools being correct.

By applying successive synthesis steps, more detailed circuit descriptions are obtained from more abstract ones. Given a circuit description, any previous description can be taken as its specification. For the synthesis process to be correct, the behavior of each new description must be included in the behavior of its predecessors. This is often referred as that the circuit *conforms to* its specification. In addition, properties that hold for previous descriptions must hold for their successive refinements as well.

In order to properly talk of "verification", a formal specification and a model of the circuit are needed. Since the original idea of what the circuit should do or be cannot be considered (almost never) as formal, the first step is normally called design *validation* rather than verification. What of course can be formally checked is a set of properties that the initial circuit specification should or should not have, e.g. properties like "the circuit can always be safely reset", or "event x is eventually acknowledged by event y".

After high-level synthesis, the obtained register-transfer level description can be checked against the behavioral description. Modules are generally assumed to be correct, but the interconnection of several modules must be verified to perform the specified operation. The control logic obtained by logic synthesis can be proven to conform to its specification. The generated layout has to correctly implement the netlist of gates obtained after logic synthesis. All the above are examples of verification that should be carried out as new circuit descriptions are available. In theory, it could be proven that the final layout conforms to the original behavioral specification, i.e. that the synthesis process has generated a correct circuit implementation from its original specification.

1.1.3 Testing

At the moment, no foundry can guarantee that any fabricated circuit is fault-free. The *theoretical yield* of a fabrication process is the percentage of *correct* or *fault-free* circuits from the total number of manufactured circuits:

$$Y_T = \frac{N_{\text{correct}}}{N_{\text{total}}} \ .$$

The quality control every circuit has to pass might be too loose or too strict... or both. In the first case some incorrect circuits may be accepted, while in the second one some correct circuits may be rejected. In fact it could happen that a portion of faulty circuits are accepted, as well as a number of fault-free ones are rejected. Since the number of correct circuits is not necessarily the number of accepted circuits, the *real yield* should be defined as

$$Y_R = \frac{N_{\text{acceptable}}}{N_{\text{total}}} \ .$$

In this case, "acceptable" means that the customer will buy the circuit thinking it is fault-free.

Test processes rejecting not only every incorrect circuit but possibly also some correct one are said to be *pessimistic* or *conservative*. On the contrary, if a test accepts all correct circuits and some faulty ones as well, it is called *optimistic*. Conservative tests are good from the customers' point of view, since they are sure of not buying any incorrect circuit. However, optimistic tests will allow manufacturers to sell a larger portion of the circuits they make. Nevertheless, it is easy to see that detecting a faulty circuit once it is normally operating is much more costly than noticing it at manufacturing time (imagine what would happen if a faulty circuit is placed in a plane and the fault manifests some seconds before landing). Taking that into account, conservative tests should be preferred to optimistic ones.

The *reject ratio* (RR) gives idea of how good a test process is. Figure 1.2 represents a set of manufactured circuits. Let N_{GP} and N_{GF} be respectively the number of good circuits that pass and fail a given test, while N_{BP} and N_{BF} are the the number of bad circuits passing and failing a test. Consequently, there are $N_{GP} + N_{GF}$ good circuits and $N_{BP} + N_{BF}$ bad ones. After applying a certain test process, only $N_{GP} + N_{BP}$ circuits pass the test, thus they are accepted as correct. The reject ratio is defined as follows:

$$RR = \frac{N_{BP}}{N_{GP} + N_{BP}} \ .$$

The smaller the reject ratio is, the tighter the test process results. Ideally, RR should be zero.

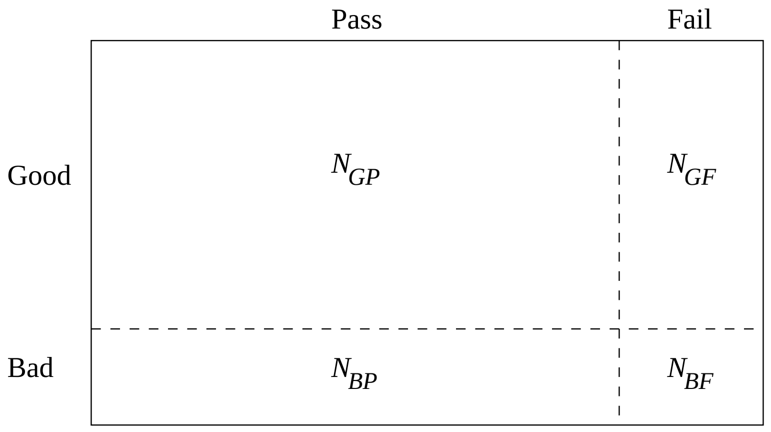


Figure 1.2 Universe of good/bad circuits given a fail/pass test.

Clearly, there are two ways of obtaining fault-free circuits: either the technology becomes capable of fabricating 100% correct circuits or testing methodologies must be able to test all possible faults in a circuit. The yield might increase as technology develops, but there is a little likelihood of having 100% correct circuits "per construction" because of physical limitations. Therefore, testing is essential in order to lower the number of incorrect circuits finally arriving to the market.

1.2 WHY ASYNCHRONOUS?

Asynchronous circuits were in fashion between late 50's and middle 60's, but since then the interest in such circuits decreased in favor of synchronous circuits. The main reason for this decline is that designing asynchronous circuits has been traditionally considered difficult. Since clocked synchronous circuits were much easier to design and did not do their work worse than their tricky counterparts, the latter were forgotten and the pioneer work by Huffman [Huf64], Muller [MB59, Mul63] and others, was covered by dust.

The simple structure of synchronous circuits, cones of combinational logic followed by clocked flip-flops, allows a very important abstraction: time becomes a discrete rather than a continuous magnitude. Therefore, the state of a circuit changes synchronously with the arrival of a clock edge. Any computation must occur between consecutive clock ticks, i.e. within the so called *clock cycle*. Whatever happens during that time is not relevant, as far as by the end of the cycle the circuit is stable. Even the major concern related to time was relatively easy to solve: searching for the slowest combinational path (the *critical path*) and assigning a long enough clock cycle in order to let it settle.

Next, the most relevant drawbacks that asynchronous circuit design has to cope with are commented:

- **Design difficulty**– The inherent concurrency of asynchronous circuits makes them difficult to design. Fortunately, automatic synthesis, verification and test pattern generation methodologies proposed in the last years are beginning to overcome this problem. In particular, the work here presented tackles the verification and testing of asynchronous circuits.
- **State explosion**– This problem refers to the fact that the number of states of a circuit with n variables can be as large as 2^n. The state in a synchronous circuit depends on its inputs and on the values of the so called *state variables*, typically a number of flip-flops. On the contrary, the state in an asynchronous circuit depends on all its signals, rather than on a subset of them. Therefore, even the state explosion problem arises both for synchronous and asynchronous circuits, the problem is harder to solve in the case of asynchronous circuits. Structural analysis, techniques taking advantage of hierarchy, partitioning the systems to make the problem affordable and using symbolic representation of the states are showing effective in mitigating the state explosion problem.
- **Larger circuits**– In general, asynchronous circuits tend to be larger than synchronous ones because of the logic needed to detect operation completion and the often more complex control logic. Also, some logic families use *dual-rail* encoding, which eases completion detection, but needs keeping two signals for each bit of information. This generally results in larger circuit area. Nevertheless, circuit area is each time a less important problem, as technology allows integrating more and more transistors in each chip.

Despite the above problems, asynchronous circuits have several potential advantages:

- **Absence of clock skew**– The *skew* of a signal is the difference between the arrival of signal changes at different points of a same signal wire. Although this skew is negligible in most cases, it becomes a major concern as the length of the wire increases. Particularly troublesome is the distribution of a global clock throughout a chip or, even worse, a whole system. Let us take as an example the 300 MHz DEC 21164 Alpha microprocessor [ea95], implemented with 0.5μm CMOS technology and featuring 1.68 million transistors, 9.3 million with the on-chip secondary cache. The clock is distributed through a ten-level binary tree, which results too large to be handled by SPICE [JQN$^+$91]. In order to simulate the clock distribution network, it has been finally partitioned into three main subsystems, each one separately simulated. After extensive SPICE simulation, the skew between the first and the last latch receiving the clock has been estimated in 90ps, which represents more than 5% of the clock cycle. The authors of [ea95] do not provide skew estimation between the clock edge entering the distribution network and arriving to the last latch, but it should be significantly larger. Moreover, as clock frequencies approach 1 GHz and interconnect pitches fall below 0.5μm, interconnect delay will become a dominant part of the clock cycle [Boh96]. Since asynchronous circuits, by definition, do not have any global synchronization signal, they are free from the clock skew problem.

- **Potential lower power consumption**– The absence of long capacitive wires, like the clock signal, that must be continuously charged and discharged provides a potential reduction in consumption. In the Alpha example above, the last clock driver has an equivalent gate width of 58cm and drives a load of 3.75nF. The clock distribution system consumes about 20W, approximately 40 % of the total chip power.

  Given an instant of time, a system component is said to be *idle* if it does not make any useful computation. In asynchronous circuits idle modules do not switch, which means, depending on the implementation technology, that those modules do not dissipate energy. Honestly, it has to be said that techniques like *clock gating* can be applied to synchronous circuits to reduce the consumption in idle modules.

- **Average-case performance**– The time to perform most computations depends on the values of the operands and not only on the operation itself. *Self-timed* modules start to operate as soon as available data is at their inputs and produce a completion event telling when the operation has finished. While synchronous pipelines must work at a *worst case* speed, self-timed pipelines, can work at a certain *average speed*, which in general will be faster than the worst case.

- **Automatic adaptation to physical changes**– The speed of circuit components may vary depending on physical changes such as temperature, power supply, variations in the manufacture process, etc. The clock cycle of a synchronous circuit must take into account the worst possible combination of all these factors. On the contrary, many asynchronous circuits sense computation completion and also some classes of asynchronous circuits properly work whatever the delay of their components is. Therefore, such circuits will automatically operate as fast as the physical characteristics allow.

- **Better technology migration potential**– Migrating from one technology to another one may be considered, in some sense, a multiple variation of a set of physical characteristics. For the same reasons exposed in the previous point, technology migration will only affect performance in most asynchronous circuits, but not correctness.

- **High degree of modularity**– Any synchronous pipeline module must be designed to satisfy the clock cycle requirements, even if it is a rarely used component. Since a synchronous pipeline operates at the speed of its slowest element, a slow unit would decrease system performance. In contrast, an infrequently used asynchronous module can be left unoptimized without significant loss in circuit performance. In a same fashion, a module can be substituted by another one (with the same interface, but possibly faster, smaller or less consuming) without further modifications to the rest of the circuit.

- **Robust mutual exclusion handling**– Even though most circuits are synchronous, their environment is inherently asynchronous. Sometimes, arbitration mechanisms are needed and, because of metastability problems, the arbiter decision may take an indefinite long time. Thus, if the arbiter's response is not be ready within the required period of time, a malfunction might be produced. Asynchronous modules can wait until the arbiter comes up with a decision with no major timing restriction.

- **Potential reduction of electro-magnetic radiation**– One of the problems of synchronous circuits is the electro-magnetic radiation as the operating frequency increases. Thousands of millions of electronic devices switching almost at a time at frequencies closer to the gigahertz are an important emitter of microwave radiation. Asynchronous transitions tend to be evenly distributed in time, therefore the absence of a proper operating frequency nearly surmounts this problem.

Time is coming that the benefits of synchronous circuits are beginning to show not as important as their drawbacks will be, whereas most of the problems of asynchronous circuits are fortunately being solved by current research. Asynchronous circuits will become a feasible design style only if their problems can be satisfactorily managed. Time and research will determine whether the potential advantages of asynchronous circuits keep forever potential or are turned into actual benefits.

1.3 ASYNCHRONOUS CIRCUIT MODELING

Asynchronous circuits can be regarded as an interconnection of two different classes of components, *gates* and *delays*, interconnected through *wires*. The differences between the two component classes are as follows:

- *Gates* are components with several inputs and one or more outputs. The value of each output is instantaneously calculated by an associated logic function that depends only on the gate inputs. Without loss of generality, we will only consider one-output gates, since n-output gates can be regarded as n one-output gates.
- *Delays* are one-input one-output elements. They reproduce at their output, after a certain amount of time, the value at their single input.

The magnitude of a delay can be either

- *bounded*, if an upper and lower bound of the delay magnitude are known, or
- *unbounded*, if no bound for the magnitude is known, except that it is positive and finite.

Depending on the "amount of memory" associated to a delay element, two major delay types, graphically represented in figure 1.3, can be considered:

- *pure*, if the delay element exactly replicates the wave at its input after the delay magnitude, and
- *inertial*, if pulses shorter than the delay magnitude are filtered out.

The place where delays are inserted introduces different ways of modeling asynchronous circuits (see figure 1.4):

- the *feedback delay* model, in which every feedback loop is cut by at least a delay element and replacing any delay by a wire produces some cycle with no delay elements,

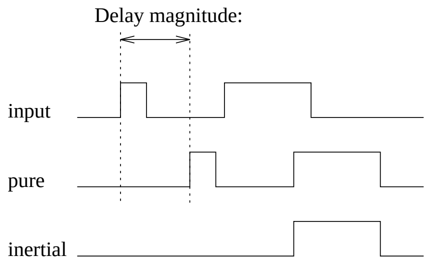


Figure 1.3 Difference between pure and inertial delay.

- the *gate delay* model, where each gate output is followed by exactly one delay element, and
- the *wire delay* model, that associates a delay to each wire. It is easy to see that this last model is equivalent to placing a delay element before each gate input.

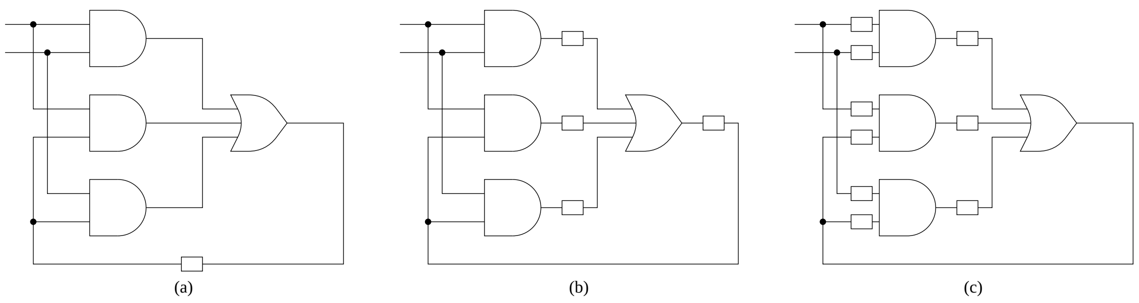


Figure 1.4 (a) Feedback, (b) gate and (c) wire delay models.

By choosing different combinations of delay magnitude, type and model, several types of asynchronous circuits can be considered. Next, the most commonly used asynchronous circuit models are described.

Huffman model

Huffman [Huf64] proposed representing an asynchronous circuit by splitting the circuit into two differentiated parts: the combinational subnetwork, following the bounded inertial wire delay model, and the feedback lines, modeled by using unbounded inertial feedback delays.

Other models have been proposed as variations of the original Huffman model, namely *fundamental mode* [Huf64], *burst mode* [Now93] and *self-clocked* [RMCF88] circuits.

Muller model (speed-independent circuits)

The model presented by Muller [MB59] associates an unbounded inertial delay element to each gate output, since the delay of the wires is considered negligible. Circuits designed following this model are also known as *speed-independent* circuits, since they correctly operate despite the delays of their components.

Delay-insensitive circuits

The most robust class of asynchronous circuits appears when circuits are modeled by using the unbounded wire delay model [MFR85]. This model is equivalent to the Muller model in those wires connecting a single gate output to a single gate input. By placing a delay before each gate input, the delay at the different ends of a *fork*, i.e. one output connected to more than one input, may vary. This is equivalent to saying that forks are not *isochronic*. Unfortunately, very few circuits made up of simple gates can operate in a delay-insensitive way.

Quasi-delay-insensitive circuits

The assumption that all forks are either isochronic or non-isochronic may be too optimistic or too pessimistic. Assuming an asynchronous circuit made up of different modules, quasi-delay-insensitive circuits [Mar90b] consider the unbounded gate delay model inside these modules, whereas their interface is modeled by using unbounded wire delays. Therefore, forks are isochronic inside a module, but they are non-isochronic outside.

Hazard-free circuits with bounded delays

Unbounded delays are a robust approximation to actual circuitry behavior, but simpler and faster circuits can be implemented if the delays of the gates are known [Lav92]. Therefore, several circuit design methodologies assume bounded delays. Such synthesis procedures must assure that no hazards, and hence no malfunction, appear when the circuit is being normally operated.

The delay of a gate belong to a certain range provided the circuit is correctly manufactured. Since the delay of gates and paths of gates must be within a certain range, testability of *delay faults* [Bre74] turns out an essential issue in circuits synthesized assuming bounded delays.

Micropipelines

Of particular interest is the work by Sutherland [Sut89] on self-timed circuits, which does not describe an actual asynchronous circuit model but a design methodology. The *micropipeline* architecture, divided into control and data-path parts, performs an asynchronous pipelined computation. Data-path elements are self-timed *bundled data* modules that communicate following a two-phase handshake. With this simple architecture several complex designs, including a fully operating asynchronous microprocessor [FDG$^+$93], have been successfully implemented.

1.4 OVERVIEW

This work tackles two key points in order to guarantee that no incorrect asynchronous circuit is issued to the market: formal verification and testing. The inherent difficulty of asynchronous circuits and the ever increasing complexity of VLSI systems require not only automated synthesis procedures, but also automated verification and test generation tools. The main contributions to be presented can be included in the processes of specification validation, circuit verification and test pattern generation. In the synthesis pipeline depicted in figure 1.1 these processes have been emphasized.

The above contributions have been implemented in form of automatic asynchronous circuit verification and test generation tools, respectively *versify* and *testify*. Both tools can be obtained via WWW from the URLs:

`http://www.ac.upc.es/recerca/VLSI/versify`
`http://www.ac.upc.es/recerca/VLSI/testify`

1.4.1 Specification validation

Several languages and formalisms allow to specify the behavior of asynchronous circuits: CSP [Hoa78, Hoa85], Petri nets [Pet62], process algebras [Ebe89], etc. We have chosen Petri nets because they are familiar to researchers in several fields, in particular to the asynchronous circuit design community.

Petri nets [Pet62] are a formalism that can describe the behavior of many asynchronous systems and, consequently, several synthesis approaches have used such nets as specification of the circuits to be implemented [RY85, Chu87]. Given a Petri net, specification validation consists of checking properties required for the net to be implementable, as well as verifying other properties that are often desirable in an asynchronous circuit specification. Because of the high parallelism that Petri nets can express, manual verification of such properties is usually unaffordable.

Chapter 3 presents how safe Petri nets can be efficiently handled by using Binary Decision Diagrams (BDDs) [Bry86]. Each *place* in a Petri net is represented as a boolean variable and each *marking* as a minterm of place variables. This allows representing sets of markings as boolean functions that can be efficiently manipulated by using BDDs. Algorithms for computing the set of reachable markings of a Petri net and verifying several properties are presented. This work has been published in [PRCB94, KCK$^+$95].

1.4.2 Asynchronous circuit verification

Chapter 4 is devised to the verification of speed-independent circuits. The verification process is performed by composing a circuit with a Petri net, which can be seen both as the circuit specification and as the circuit environment. In this closed system, some transitions in the environment are associated to some input or output circuit signals. The environment "decides" when the circuit inputs must switch, whereas when the circuit switches some output signal, the associated transition in the environment fires accordingly.

The state of a speed-independent circuit can be represented as a minterm, provided each circuit signal is associated to a boolean variable. Therefore, the state of an environment-circuit system can also be modeled with BDDs. Two different verification approaches are presented: the first one uses a flat netlist description of the circuit, while the second one takes benefit of the abstraction of many internal combinational circuit signals. These two techniques have been published respectively in [RPC94, RCP95b] and [RCP95a].

1.4.3 Testing of asynchronous circuits

The last contribution tackles Automatic Test Pattern Generation (ATPG) for asynchronous circuits. Since commercial test equipment is inherently synchronous, a technique for testing asynchronous circuits by using synchronous test patterns is proposed. When such patterns are applied to an asynchronous circuit, two undesired situations may arise: *oscillation* and *non-confluence*. In the first case the circuit never stabilizes, while in the second one, depending on the delays of the circuit components, the circuit may settle in different states.

In chapter 5 the behavior of asynchronous circuits under test is modeled. From that model, all the possible input patterns that cause either oscillation or non-confluence are eliminated. Finally, an ATPG algorithm obtains a set of patterns in order to cover single stuck-at faults. This testing methodology is to appear in [RCPP97].

1.5 STRUCTURE OF THE WORK

Chapter 2 outlines the mathematical basis for boolean reasoning. Chapters 3 and 4 are devoted to formal verification, respectively to specification validation and circuit verification. The testing approach for asynchronous circuits is presented in chapter 5. Finally, conclusions and future lines of work are pointed out in chapter 6.

2

MATHEMATICAL BACKGROUND

As far as the laws of mathematics refer to reality, they are not certain, and as far as they are certain, they do not refer to reality.

– Albert Einstein.

The good Christian should beware of mathematicians and all those who make empty prophecies. The danger already exists that mathematicians have made a covenant with the devil to darken the spirit and confine man in the bonds of Hell.

– St. Augustine.

About this chapter

This chapter introduces some mathematical concepts and notation that will be used in the following chapters. How some of these concepts can be manipulated by a computer is also explained in a section devoted to *Binary Decision Diagrams* (BDDs). The notation has been borrowed, in part, from [Bro90, BS95].

Since this is intended as an introduction, readers familiar with the notation of sets, algebras and BDDs can skip reading this chapter.

2.1 PRELIMINARIES

In this section some very basic concepts and notation appearing in formulae are presented.

A *proposition* is a formula that is either true or false. Examples of propositions are:

- $3 \leq 2$.
- 2 is a prime number.
- $(\forall x)[x \in \emptyset \Rightarrow x \in \{1,2\}]$.

The formula

$$x + y \leq 2$$

is not a proposition, since it is neither true nor false. It would be a proposition if x and y were known to be some particular numbers. Such a formula is called a *predicate*. More formally, a predicate $P(x_1, \ldots, x_n)$ is a formula involving the variables x_1 to x_n. Note that an n-variable predicate becomes a proposition for each possible substitution of values for its n variables. Thus, a proposition can be seen as a 0-variable predicate.

Several symbols may be used in formulae to combine different predicates or to quantify their variables. Whether a symbol or its expression in natural language is used in a formula will depend on the readability of each particular formula. Next some of these symbols and their usual meaning are presented:

- $\forall$: for all ...
- $\exists$: for at least one ...
- $\Rightarrow$: implies ...
- $\Leftrightarrow$: equivalence ...
- $\wedge$: and ...
- $\vee$: or ...
- $\neg$: not ...

2.2 SETS, RELATIONS AND FUNCTIONS

A *set* is a collection of distinct objects, usually called *elements* or *members*. The fact that an element a belongs (does not belong) to a set $\mathcal{A}$ is written $a \in \mathcal{A}$ ($a \notin \mathcal{A}$). All the elements in a set are taken from a *universe of discourse* or *universal set*. Commonly universal sets are written in calligraphic letters ($\mathcal{A}$, $\mathcal{R}$, $\mathcal{U}$, $\mathcal{Z}$, etc), whereas sets within them are written in capital italic letters (A, B, E, S, etc). Sets can be described in different ways:

- By listing its elements between curly braces — $\{a, e, i, o, u\}$, $\{e_1, \ldots e_n\}$, etc.
- By expressing the elements from the universal set that have some property — $\{z \in \mathcal{Z} | z < 10\}$, $\{x \in \text{“phonemes”} | x \text{ is a vowel}\}$, etc.
- By using a recursive rule like the following:
  1. *BASE*: a list of elements belonging to the set.
  2. *RECURSION*: a construction-rule describing how to generate new members of the set from other members of the set.
  3. *RESTRICTION*: a statement, often omitted, saying that the only set members are those described either in the base or in the recursion.

The *cardinality* of a set A, denoted as $|A|$, is the number of elements in the set. The *empty set*, written $\emptyset$, is a set with cardinality zero.

Given two sets A and B from the same universe of discourse $\mathcal{U}$, different predicates and operations can be defined:

- $A = B$ (*equality*): Two sets are equal iff they contain exactly the same elements. If they differ it is written $A \neq B$.

- $A \subseteq B$ (*inclusion*): Set A is included in set B iff all the elements of A are also in B. If some element of A is not in B we can write $A \nsubseteq B$.
- $A \subset B$ (*proper inclusion*): Set A is properly included in B iff $A \subseteq B$ and $A \neq B$. Otherwise it is denoted as $A \not\subset B$.
- $\bar{A}$ (*complementation*): The complement of A is the set of all elements in $\mathcal{U}$ that are not in A. Note that if $A = \emptyset$, then $\bar{A} = \mathcal{U}$, and vice versa.
- $A \cup B$ (*union*): The union of two sets is the set of all elements that appear in either A or B.
- $A \cap B$ (*intersection*): The intersection of two sets is the set of all elements in both A and B.
- $A - B$ (*difference*): The difference of two sets A and B can be defined as $A \cap \bar{B}$.

Given two sets A and B, not necessarily from the same universal set, the *Cartesian product* $A \times B$ is defined by the following set of ordered pairs:

$$A \times B = \{(a,b) | a \in A \text{ and } b \in B\} \ .$$

This definition can be easily extended to more than two sets. The Cartesian product of n sets $A_1, \ldots A_n$ can be defined as the next set of ordered n-tuples:

$$A_1 \times \ldots \times A_n = \{(a_1, \ldots, a_n) | a_i \in A_i \text{ for } 1 \leq i \leq n\} \ .$$

If all n sets are the same set, e.g. A, the Cartesian product is often abbreviated as A^n. For example, $A \times A$ is A^2, $A \times A \times A$ is A^3, etc.

The *power set* of a set A, written 2^A, is defined as the set of all subsets of A:

$$2^A = \{B | B \subseteq A\} \ .$$

This notation comes from the fact that a set with n elements has 2^n possible subsets. In other words, $|2^A| = 2^{|A|}$.

A *binary relation* (often just *relation*) R from set A to set B is a subset of their Cartesian product ($R \subseteq A \times B$). If $a \in A$ is in relation R with $b \in B$, it can denoted either by $(a,b) \in R$ or by aRb. A *relation on A* is a relation from A to itself. Given a relation R on A, the *composition* of R with R, written RR or R^2, is defined as

$$R^2 = \{(a,c) \in A^2 | \exists b \in A \text{ with } aRb \text{ and } bRc\}$$

Considering that $R^1 = R$ and $R^{n+1} = RR^n$ for $n \geq 1$, the *transitive closure* of R, denoted R^+, can be defined as

$$R^+ = \{(a,b) \in A^2 | aR^k b \text{ for some } k \geq 1\}$$

The *reflexive and transitive closure* of R is defined as

$$R^* = \mathrm{id}_A \cup R^+ \ .$$

Given any relation R on A, it is said to be:

- *Reflexive* if aRa for all $a \in A$.
- *Symmetric* if aRb implies bRa for all $a, b \in A$.
- *Antisymmetric* if aRb and bRa implies $a = b$ for all $a, b \in A$.

- *Transitive* if aRb and bRc implies aRc for all $a, b, c \in A$.

The relation $\mathrm{id}_A = \{(a,a) | a \in A\}$ is called the *identity relation*, and relates each element with itself.

A relation on a set A that is reflexive, symmetric and transitive is an *equivalence relation* on A. A *partial order* on A is a reflexive, antisymmetric and transitive relation on a A.

Given a set A, a partial order $\leq$ on A, and a nonempty subset S of A, the element $u \in A$ is called an *upper bound* of S if $s \leq u$ for all $s \in S$. An upper bound u of S is said to be the *least upper bound* of S, written $lub(S)$, if $u \leq b$ for all b upper bound of S. An example of least upper bound can be found at the end of section 2.5.

A *function* from a set A into a set B, written $f : A \longrightarrow B$, is a rule that relates each element $a \in A$ to one element $b = f(a) \in B$ called the *image* of a. A function is a particular case of relation, where each element in A appears only in one pair of the relation. The sets A and B are respectively called the *domain* and the *codomain* of the function. A function is said to be *one-to-one* iff $x \neq y$ implies $f(x) \neq f(y)$.

An *n-variable function* from A into B is a function of the form

$$f : A^n \longrightarrow B \ .$$

Given a function $f : A \longrightarrow B$ and a set $C \subseteq A$, the set

$$\mathrm{Img}(f, C) = \{b \in B | \exists a \in C \text{ such that } b = f(a)\}$$

is called the *image of set C under f*. Symmetrically, given the same function and the set $I \subseteq B$, the set

$$\mathrm{Pre}(f, I) = \{a \in A | \exists b \in I \text{ such that } b = f(a)\}$$

is called the *preimage of set I under f*.

2.3 LOGIC FUNCTIONS

The set $\{0, 1\}$ is commonly denoted as B (not to be confused with a generic set B used in previous sections). An *n-variable logic function* (also *boolean* or *switching function*) is a mapping

$$f : \mathrm{B}^{\mathrm{n}} \longrightarrow \mathrm{B} \ .$$

Each element of B^{n} is also called a *vertex*. The *on-set* (*off-set*) of a function f is the set of vertices where the function evaluates to 1 (0). Each vertex of the on-set is also called a *minterm*. A *literal* is either a variable or its complement. A *cube* c is a set of literals, such that if $x_i \in c$ then $\bar{x}_i \notin c$ and vice versa. A cube is interpreted as the boolean product of its elements. The cubes with n literals are in one-to-one correspondence with the vertices of B^{n}.

The functions

$$f\downarrow_{x_i} = f(x_1, \ldots, x_{i-1}, 1, x_{i+1}, \ldots, x_n)$$

and

$$f\downarrow_{\bar{x}_i} = f(x_1, \ldots, x_{i-1}, 0, x_{i+1}, \ldots, x_n)$$

are called the *cofactor* of f with respect to x_i and $\bar{x}_i$ respectively. The definition of cofactor can also be extended to cubes. If $c = x_1c_1$, x_1 being a literal and c_1 another cube, then

$$f\downarrow_c = (f\downarrow_{x_1})\downarrow_{c_1} \quad .$$

Abstractions are of fundamental use in our framework. As it will be shown later, they have a direct correspondence to the existential and universal quantifiers applied to predicates in boolean reasoning. The *existential* and *universal abstractions* of f with respect to x_i are defined as:

$$\exists_{x_i} f = f\downarrow_{x_i} + f\downarrow_{\bar{x}_i} \quad ; \qquad \forall_{x_i} f = f\downarrow_{x_i} \cdot f\downarrow_{\bar{x}_i} \quad .$$

As an example, let us consider the function

$$f = bc + a\bar{b}\bar{c} + \bar{a}c \quad .$$

The cofactors with respect to a and $\bar{a}$ are

$$f\downarrow_a = bc + \bar{b}\bar{c} \quad \text{and} \quad f\downarrow_{\bar{a}} = c \quad ,$$

and the abstractions with respect to a are

$$\exists_a f = f\downarrow_a + f\downarrow_{\bar{a}} = \bar{b} + c \quad \text{and} \quad \forall_a f = f\downarrow_a \cdot f\downarrow_{\bar{a}} = bc \quad .$$

The function $\exists_a f$ evaluates to 1 for all those values of b and c such that there is a value of a for which f evaluates to 1. The function $\forall_a f$ evaluates to 1 for all those values of b and c such that f evaluates to 1 for any value of a.

2.4 BOOLEAN ALGEBRA

An *algebraic system* (or simply *algebra*) is a set, called the *carrier*, together with one or more operations over the carrier that satisfy specific axioms. A *Boolean algebra* is an algebraic system

$$\langle B, +, \cdot, 0, 1 \rangle$$

in which B is the carrier, $+$ and $\cdot$ (respectively referred as *addition* and *product*) are binary operations on B, and 0 and 1 are distinct elements of B. Note that B may be any set, not necessarily B $= \{0, 1\}$. This algebra satisfies the following postulates:

1. Commutative laws. For all $x, y \in B$:
$$x + y = y + x$$
$$x \cdot y = y \cdot x$$
2. Distributive laws. For all $x, y, z \in B$:
$$x + (y \cdot z) = (x + y) \cdot (x + z)$$
$$x \cdot (y + z) = (x \cdot y) + (x \cdot z)$$
3. Identities. For all $x \in B$:
$$x + 0 = x$$
$$x \cdot 1 = x$$
4. Complements. For all $x \in B$ exists a unique element $\bar{x} \in B$ such that
$$x + \bar{x} = 1$$
$$x \cdot \bar{x} = 0$$

As in ordinary algebra, the product symbol "$\cdot$" is often omitted, and the product takes precedence over the addition. Thus the formulae $x + (y \cdot z)$ and $x + yz$ are equivalent.

2.4.1 Switching algebra

The smallest Boolean algebra is the system

$$\langle \mathrm{B}, +, \cdot, 0, 1\rangle \ ,$$

where $+$ and $\cdot$ are respectively defined as the logic OR and the logic AND, and the complement as the logic NOT, i.e.

x	y	$x+y$
0	0	0
0	1	1
1	0	1
1	1	1

x	y	$x \cdot y$
0	0	0
0	1	0
1	0	0
1	1	1

x	$\bar{x}$
0	1
1	0

This system is known as the *switching algebra* and sometimes it is referred simply as *the* algebra.

2.4.2 Algebra of classes (subsets of a set)

A boolean algebra of particular interest is the *algebra of classes*. In many situations it is interesting dealing with the subsets of a given universal set $\mathcal{U}$. In this case the different subsets of $\mathcal{U}$ are called the *classes* of $\mathcal{U}$. The *algebra of classes* consists of the set $2^{\mathcal{U}}$ (the power set of $\mathcal{U}$), the operations $\cup$ and $\cap$ (respectively the union and intersection of sets), and the distinct elements $\emptyset$ and $\mathcal{U}$. The algebraic system

$$\langle 2^{\mathcal{U}}, \cup, \cap, \emptyset, \mathcal{U}\rangle$$

satisfies the postulates of Boolean algebra.

2.4.3 Algebra of logic functions

Let $F_n(\mathrm{B})$ be the set of n-variable logic functions on B. Then the system

$$(F_n(\mathrm{B}), +, \cdot, 0, 1) \ ,$$

is also a boolean algebra, in which $+$ and $\cdot$ signify addition and multiplication of logic functions, and 0 and 1 signify the zero- and one-functions ($f(x_1, \ldots, x_n) \equiv 0$ and $f(x_1, \ldots, x_n) \equiv 1$). The cardinality of $F_n(\mathrm{B})$ (number of different n-variable logic functions) is 2^{2^n}.

2.4.4 Stone's representation theorem

Next, the *Representation Theorem* (Stone, 1936) establishes one of the basis that will allow to symbolically represent sets.

Theorem 2.1 (Stone, 1936) *Every finite boolean algebra is isomorphic to the boolean algebra of subsets of some finite set S.*

Consequently, Stone's theorem states that reasoning in terms of concepts such as *union*, *intersection*, *empty set*, etc, in a finite set of elements is isomorphic to performing logic operations

$(+, \cdot)$ with logic functions. Furthermore, from Stone's theorem it can be easily deduced that the cardinality of the carrier of any boolean algebra must be a power of two. In particular, the algebra of classes of a set S such that $|S| = 2^n$, is isomorphic to the boolean algebra of n-variable logic functions.

2.5 TERNARY ALGEBRA

Sometimes, when analyzing digital circuits it is not enough working with the two logical values 0 and 1. A third value, Φ, standing for an uncertain value which is neither 0 nor 1 might be also needed. A *ternary algebra* is an algebraic system

$$\langle T, +, \cdot, 0, \Phi, 1\rangle$$

where T is a set, $+$ and $\cdot$ are binary operations on T, and 0, Φ, and 1 are distinct elements of T. A ternary algebra resembles a boolean algebra, but it is *not* a boolean algebra. The first three postulates of boolean algebra, commutativity, associativity and identity are the same for ternary algebra. However, the postulates concerning complements, $x + \bar{x} = 1$ and $x\bar{x} = 0$ do not hold when $x = \Phi$, since $\Phi = \bar{\Phi}$.

Let us call T the set $\{0, \Phi, 1\}$. The smallest ternary algebra is the system $\langle \mathrm{T}, +, \cdot, 0, \Phi, 1\rangle$, where the product, addition and complement are described as follows:

		y		
	$x+y$	0	Φ	1
	0	0	Φ	1
x	Φ	Φ	Φ	1
	1	1	1	1

		y		
	$x \cdot y$	0	Φ	1
	0	0	0	0
x	Φ	0	Φ	Φ
	1	0	Φ	1

x	$\bar{x}$
0	1
Φ	Φ
1	0

The "uncertainty" partial order, $\sqsubseteq$, on T is defined as follows:

$$0 \sqsubseteq 0,\;\; 1 \sqsubseteq 1,\;\; \Phi \sqsubseteq \Phi,\;\; 0 \sqsubseteq \Phi,\;\; 1 \sqsubseteq \Phi,$$

and no other pairs are related by $\sqsubseteq$. Thus, if $x, y \in \mathrm{T}$, $x \sqsubseteq y$ can be read like x "is at least as uncertain as" y. Considering the above definition, it is easy to see that $lub(\{0\}) = 0$, $lub(\{1\}) = 1$, and $lub(\{0,1\}) = lub(\{0,\Phi\}) = lub(\{1,\Phi\}) = lub(\{\Phi\}) = lub(\{0,\Phi,1\}) = \Phi$. The least upper bound for any two elements $x, y \in \mathrm{T}$ can be computed as

$$lub(\{x, y\}) = a \cdot b + (a + b) \cdot \Phi \;\; .$$

The uncertainty partial order, $\sqsubseteq$, can be naturally extended for $\mathrm{T^n}$. Let us consider $x = (x_1, \ldots, x_n)$ and $y = (y_1, \ldots, y_n)$, with $x, y \in \mathrm{T^n}$. If $x_i \sqsubseteq y_i$ for all i, $1 \le i \le n$, then $x \sqsubseteq y$. The least upper bound can also be extended in the same way. In particular, given any two elements $x, y \in \mathrm{T^n}$,

$$lub(\{x, y\}) = \big(lub(\{x_1, y_1\}), \ldots, lub(\{x_n, y_n\})\big) \;\; .$$

2.6 MAKING IT EFFICIENTLY: BDDS

A logic function can be represented in many ways, such as *truth tables*, *Karnaugh maps* or *minterm canonical forms*. Another form that can be much more compact is the *sum of products*, where the logic function is represented by means of an equation, i.e,

$$f = bc + a\bar{b}\bar{c} + \bar{a}c \ . \tag{2.1}$$

These techniques are inefficient for fairly large functions. However, all these forms can be canonical [Bro90]. A form is canonical, if the representation of any function in that form is unique. Canonical forms are useful for verification techniques, because equivalence test between functions is easily computable.

Recently, *Binary Decision Diagrams* (BDDs) have emerged as an efficient canonical form to manipulate large logic functions. The introduction of BDDs is relatively old [Lee59], but only after the recent work of Bryant [Bry86] they transformed into a useful tool. For a good review on BDDs we refer to [Bry86, BBR90, TSL$^+$90, Min96].

We will present BDDs by means of an example. Given equation (2.1), its BDD is shown in figure 2.1(a). A BDD is a *Directed Acyclic Graph* with one root and two leaf nodes (0 and 1). Each node has two outgoing arcs labeled T (*then*) and E (*else*). To evaluate f for the assignment of variables $a = 1$, $b = 0$, and $c = 1$, we only have to follow the corresponding directed arcs from the root node. The first node we encounter is labeled with variable a, whose value is 1. Given this assignment, the T arc must be taken. Next, a node labeled with variable b is found. Since $b = 0$ the E arc must be taken now. Finally the T arc for variable c reaches the 0 leave node.

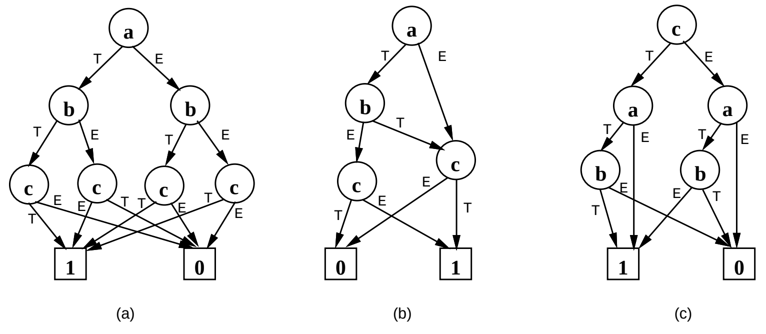


Figure 2.1 BDD example.

2.6.1 Reduced and ordered BDDs

The representation of a function by means of a BDD is not unique. Figures 2.1(a), 2.1(b) and 2.1(c) depict different BDDs representing equation (2.1). The BDD in figure 2.1(b) can be obtained from figure 2.1(a) by successively applying reduction rules that eliminate *isomorphic subgraphs* from the representation [Bry86]. The BDD in figure 2.1(c) has a different variable ordering.

All BDDs shown in figure 2.1 are *ordered* BDDs. In an ordered BDD, all variables appear in the same order along all paths from the root to the leaves. If a BDD is *ordered* and *reduced* (no further reductions can be applied) then we have a reduced and ordered BDD (ROBDD). Given a total ordering of variables, an ROBDD is a canonical form [Bry86]. Figures 2.1(b) and 2.1(c) are ROBDDs with variable ordering $a < b < c$ and $c < a < b$ respectively. The shape and size of an ROBDD highly depend on the ordering of its variables.

Some important properties of ROBDDs are:

- The size of the BDD can be exponential in the number of variables [LL92]; however BDDs are a compact representation for many functions.

- Boolean binary operations can be calculated in polynomial time in the size of the BDDs.
- Some interesting problems like satisfiability, tautology and complementation can be solved in constant time using BDDs.

Henceforth, we will implicitly assume that BDDs are reduced and ordered. Note that each BDD node represents at the same time a function whose root is the node itself. This property allows the implementation of BDD packages managing all BDDs using the same set of variables in only one multi-rooted graph [BBR90].

2.6.2 Boole's expansion theorem

The theorem given below is the basis for computation with boolean functions. Although it is frequently attributed to Shannon [Sha49], it was in fact discussed one century before by Boole [Boo54].

Theorem 2.2 (Boole's expansion) *If* $f : \mathrm{B}^{\mathrm{n}} \longrightarrow \mathrm{B}$ *is a boolean function, then*

$$f(x_1, \ldots, x_i, \ldots, x_n) = x_i \cdot f\!\downarrow_{x_i} + \bar{x}_i \cdot f\!\downarrow_{\bar{x}_i} \quad .$$

for all $(x_1, \ldots, x_i, \ldots, x_n) \in \mathrm{B}^{\mathrm{n}}$.

2.6.3 Boolean operations with ROBDDs

Let us see how to calculate the BDD for equation (2.1) given the ordering $a < b < c$. We will use (v, T, E) to denote a node labeled with variable v, and T and E as "Then" and "Else" BDDs respectively. Applying Boole's theorem to expand f with variable a we have:

$$f = a\ f\!\downarrow_a + \bar{a}\ f\!\downarrow_{\bar{a}} \quad ,$$

with $f\!\downarrow_a = bc + \bar{b}\bar{c}$, and $f\!\downarrow_{\bar{a}} = c$. Expanding variable b in $f\!\downarrow_a$ and $f\!\downarrow_{\bar{a}}$ yields to

$$f = a\ (b\ f\!\downarrow_{ab} + \bar{b}\ f\!\downarrow_{a\bar{b}}) + \ \bar{a}\ (b\ f\!\downarrow_{\bar{a}b} + \bar{b}\ f\!\downarrow_{\bar{a}\bar{b}})$$

with $f\!\downarrow_{ab} = c$, $f\!\downarrow_{a\bar{b}} = \bar{c}$, $f\!\downarrow_{\bar{a}b} = c$, and $f\!\downarrow_{\bar{a}\bar{b}} = c$. Thus the BDD for equation (2.1) is

$$f \ = \ (a, (b, c, \bar{c}), (b, c, c)) \quad .$$

Note that the logic functions $f\!\downarrow_{ab} = f\!\downarrow_{\bar{a}b} = c$ and $f\!\downarrow_{a\bar{b}} = f\!\downarrow_{\bar{a}\bar{b}} = \bar{c}$ are isomorphic and must be represented with the same node if we want to preserve canonicity.

BDDs can be created by combining existing BDDs by means of boolean operations like AND, OR, and XOR. This approach is implemented using the *if-then-else* operator (`ite`), defined as follows:

$$\texttt{ite}(F, G, H) = F \cdot G + \overline{F} \cdot H \quad ,$$

where F, G, H are logic functions represented by BDDs. The interesting property of the `ite` operator is that it can directly implement all two-operand logic functions. For example:

$$\mathrm{AND}(F, G) = \texttt{ite}(F, G, 0) \ , \quad \mathrm{XOR}(F, G) = \texttt{ite}(F, \overline{G}, G) \quad .$$

Let $Z = \texttt{ite}(F, G, H)$, and let v be the *top* variable of F, G, H. Then the BDD for Z is recursively computed as follows [Bry86]:

$$Z = (v, \texttt{ite}(F\!\downarrow_v, G\!\downarrow_v, H\!\downarrow_v), \texttt{ite}(F\!\downarrow_{\bar{v}}, G\!\downarrow_{\bar{v}}, H\!\downarrow_{\bar{v}})) \quad ,$$

```
ite(F, G, H) {
   if ( terminal case ) return result for terminal case;
   else if ( {F, G, H} is in computed table )
      return pre-computed result;
   else {
      let $v$ be the top variable of { F, G, H };
      T = ite($F\downarrow_v$,$G\downarrow_v$,$H\downarrow_v$);
      E = ite($F\downarrow_{\bar{v}}$,$G\downarrow_{\bar{v}}$,$H\downarrow_{\bar{v}}$);
      if T equals E return T;
      R = find_or_add_unique_table($v$,T,E);
      insert_computed_table({ F, G, H }, R);
      return R;
} }
```

Figure 2.2 The ite algorithm.

where the terminal cases are:

$$\texttt{ite}(1, F, G) = \texttt{ite}(0, G, F) = \texttt{ite}(F, 1, 0) = \texttt{ite}(G, F, F) = F \ .$$

The code for the `ite` algorithm is shown in figure 2.2. Note that the algorithm keeps the BDD reduced by checking if T equals E, and checking in a *unique-table* if the produced node already exists in the graph. In this way, all isomorphic subgraphs are always eliminated.

Unless there is a terminal case, every call to the procedure generates two other calls, so the total number of `ite` calls would be exponential in the number of variables. To avoid this exponentiality, `ite` uses a table of precomputed operations (*computed table*). The *computed table* acts as a cache memory, in such a way that the most recently used results are stored in this table. The effect of this computed table is to cause `ite` to be called at most once for each possible combination of the nodes in F, G, H. So the complexity of the algorithm, under the assumptions of infinite memory and constant access time (hash) tables, is reduced to $O(|F| \cdot |G| \cdot |H|)$.

An important consequence of representing all BDDs in the same graph is that checking the equivalence between two BDDs can be done in constant time (two BDDs representing the same function have the same root node). Counting the number of vertices represented by a BDD can be done in linear time in the size of the BDD.

3
SPECIFICATION VALIDATION

'And a Maker's Mark for me,' she added, pointing to a bottle high on the bar shelf.

*A proper drinker, I noted with approval. Your experienced lapper knows that barmen always initially mishear the name of whatever brand you specify. 'Not Glenlivet, Glen*fiddich*! No, you oaf, not a lager shandy, a* large brandy*... ' Always find the bottle with your eye first and point at it when ordering. Saves time.*

– Stephen Fry, "The Hippopotamus".

About this chapter

This chapter is devoted to the analysis of Petri nets, since they can be used as specification of asynchronous circuits. By using symbolic BDD-based techniques, the set of reachable Petri net markings is computed. This set is subsequently used to validate the Petri net, i.e. to verify diverse Petri net properties. Finally, the soundness of the approach is illustrated by verifying distinct benchmarks.

3.1 INTRODUCTION

The goal of specification validation is to prove that some properties of a circuit specification hold. Circuit behavior is often described by using a high level formalism, e.g. Hardware Description Languages or Petri nets. In particular, several asynchronous circuit synthesis methodologies use Petri nets as circuit specification [RY85, Chu87, Van90, Lav92, Pas96]. We will focus the attention on Petri nets, since they are used in our design framework.

A Petri net is a mathematical model of a parallel system, capable of representing causality between events. Following the definition given in [DE95], a Petri net has two components, the *net* and the *initial marking*. The net describes the relations between events, while the initial marking can be seen as the "initial state" of the Petri net. The net is a graph with two types of nodes: *places* and *transitions*. The graph is bipartite as arcs can only go from a place to a transition or vice versa. Graphically, places are represented by circles and transitions by boxes or bars.

The idea of "state" of a net corresponds to the term *marking*. A marking is a distribution of tokens in several places of the net. Tokens are graphically represented as black dots inside places. A place is said to be *marked* if it holds some token. Transitions are related with the idea of events. A transition t is said to be *enabled* if each predecessor place of t has at least one token. *Firing* transition t, i.e. letting the event occur, removes one token from each predecessor place and adds one token to each successor place. Firing a transition changes the marking of the net.

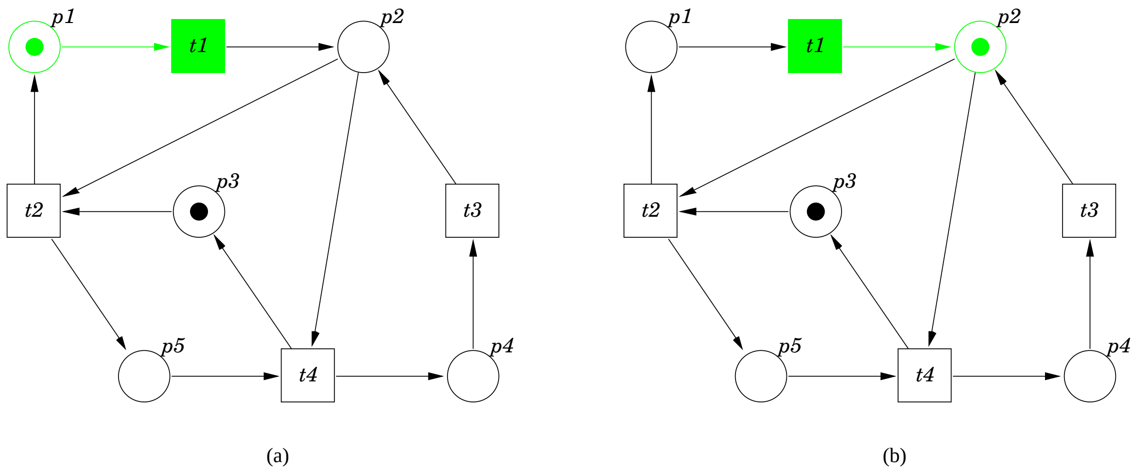


Figure 3.1 (a) A sample Petri net. (b) The same net after firing transition t_1.

Figure 3.1 illustrates a Petri net. In figure 3.1(a) place p_1 is marked, therefore transition t_1 is enabled. The token from place p_1 is removed, transition t_1 occurs and, consequently, place p_2 becomes marked. Figure 3.1(b) shows the same net after firing transition t_1.

3.1.1 Previous work

The analysis methods for Petri nets can be divided into four different groups [Sil85]:

1. **Enumeration methods**: These techniques are limited to bounded Petri nets, as they are based in the exhaustive generation of all reachable markings of a Petri net. With that information different properties can be checked straightforwardly. The effectiveness of enumeration analysis approaches will depend on their ability of managing a huge number of markings.

2. **Structural methods**: In order to avoid the state explosion problem that appears with the enumeration techniques, the structural analysis can prove different Petri net properties by studying the structure of the net. Such methods are, in general, very efficient, but they cannot be used with all types of Petri nets.

3. **Transformation methods**: In many cases a Petri net can be changed into another Petri net by applying certain transformations that preserve the properties to be checked. The objective of this transformation is obtaining a Petri net such that those properties can be verified more efficiently than in the original net. A particular case of transformation techniques are *reduction* methods, that obtain smaller Petri nets by using certain transformations.

4. **Simulation methods**: This last group of methods differ from the previous in the sense that they do not prove but give some degree of confidence that a Petri net has a given property. However, they are good at dealing with timed Petri nets or at estimating the response of systems also described by simulation.

These methods are not excluding but complementary. In practice, most authors combine different techniques to obtain hybrid approaches more efficient in solving particular problems. The interested reader can find more information about Petri nets elsewhere, e.g. [Pet62, Pet81, Sil85, Mur89, DE95].

3.1.2 Overview

Our approach combines enumeration and reduction techniques. The state explosion problem is reduced by representing sets of markings with BDDs and by simulating firing rules also with BDDs. Several reduction transformations are applied in order to reduce the complexity of Petri nets, thus obtaining a more efficient method.

The approach is further extended for Signal Transition Graphs (STGs), an interpreted type of Petri net in which transitions represent signal transitions. STGs are used by several asynchronous circuit synthesis methodologies as circuit specification, and will also be used in chapter 4 for verification of circuits.

The approaches by Hamaguchi et al. [HHY92] and Yoneda et al. [YHTM96], which can be included in the first group of methods, are particularly related to the work to be presented. Both authors manage Petri nets with BDDs to mitigate the state explosion problem and use temporal logic to verify Petri net properties.

In section 3.8 algorithms for verifying both Petri net a STG properties are described. The results obtained by applying our approach to diverse benchmarks can be found in section 3.9.

3.2 PETRI NETS

Formally, a Petri net is a quadruple $\langle \mathcal{P}, \mathcal{T}, \mathcal{F}, m_0 \rangle$, where

$\mathcal{P} = \{p_1, p_2, \ldots, p_n\}$ is a finite set of places,
$\mathcal{T} = \{t_1, t_2, \ldots, t_m\}$ is a finite set of transitions, satisfying $\mathcal{P} \cap \mathcal{T} = \emptyset$ and $\mathcal{P} \cup \mathcal{T} \neq \emptyset$,
$\mathcal{F} \subseteq (\mathcal{P} \times \mathcal{T}) \cup (\mathcal{T} \times \mathcal{P})$ is a set of arcs, also called the *flow relation*, and
m_0 is the initial marking.

The symbols ${}^\bullet t$ and $t^\bullet$ respectively define the sets of predecessor and successor places of transition t, while ${}^\bullet p$ and $p^\bullet$ are the sets of predecessor and successor transitions of place p, respectively.

A *marking* of a Petri net is an assignment of a nonnegative integer, representing a number of tokens, to each place. If k is assigned to place p, we will say that p is marked with k tokens. The flow relation of a Petri net defines a set of firing rules that determine the behavior of the net. A transition t is enabled when each $p \in {}^\bullet t$ has at least one token. The Petri net moves from one marking to another by firing one of the enabled transitions. When a transition t fires, one token is removed from each place $p \in {}^\bullet t$ and one token is added to each place $p \in t^\bullet$.

If m and m' are markings, $m[t\rangle m'$ denotes the fact that m' is reached from m by firing transition t. A marking m' is *reachable* from a marking m if there exists a sequence of transition firings that transforms m into m'. The set of reachable markings from m is denoted by $[m\rangle$. Therefore, the set of reachable markings of a Petri net $\langle \mathcal{P}, \mathcal{T}, \mathcal{F}, m_0 \rangle$ is written as $[m_0\rangle$.

Next, different types of Petri nets are defined depending on the maximum number of tokens that a place holds in any marking belonging to $[m_0\rangle$.

Definition 3.1 (Bounded Petri net) *A Petri net $\langle \mathcal{P}, \mathcal{T}, \mathcal{F}, m_0 \rangle$ is said to be* bounded *if $[m_0\rangle$ is a finite set.*

Definition 3.2 (k-bounded Petri net) *A Petri net $\langle \mathcal{P}, \mathcal{T}, \mathcal{F}, m_0 \rangle$ is said to be* k-bounded *if no place $p \in \mathcal{P}$ holds more than k tokens for any marking $m \in [m_0\rangle$.*

Definition 3.3 (Safe Petri net) *A Petri net is said to be* safe *if it is 1-bounded.*

3.2.1 Modeling safe Petri nets

Let $\langle \mathcal{P}, \mathcal{T}, \mathcal{F}, m_0 \rangle$ be a safe Petri net. A marking in $[m_0\rangle$ can be represented by a set of places m, where $p_i \in m$ denotes the fact that there is a token in p_i. Therefore, any set of markings in $[m_0\rangle$ can be represented by a set M of subsets of $\mathcal{P}$. Let $\mathcal{M}_\mathcal{P}$ be the set of all markings of a safe Petri net with $|\mathcal{P}|$ places ($|\mathcal{M}_\mathcal{P}| = 2^{|\mathcal{P}|}$). Then the system

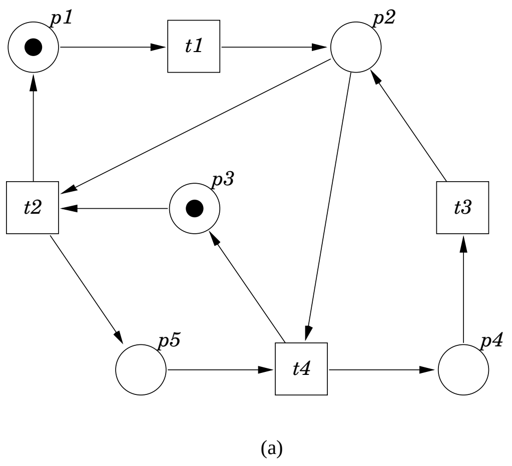
$$(2^{\mathcal{M}_\mathcal{P}}, \cup, \cap, \emptyset, \mathcal{M}_\mathcal{P})$$

is the boolean algebra of sets of markings. By Stone's theorem (see section 2.4.4), this system is isomorphic to the boolean algebra of n-variable logic functions, being $n = |\mathcal{P}|$.

We will indistinctly use p_i to denote a place in $\mathcal{P}$ or a variable in the boolean algebra of n-variable logic functions. Therefore, there is a one-to-one correspondence between markings of $\mathcal{M}_\mathcal{P}$ and vertices of B^n. A marking $m \in \mathcal{M}_\mathcal{P}$ is represented by means of an *encoding function* that provides a binary mapping from $\mathcal{M}_\mathcal{P}$ into B^n, that is, $\mathcal{E} : \mathcal{M}_\mathcal{P} \rightarrow \mathrm{B}^\mathrm{n}$, where the image of a marking $m \in \mathcal{M}_\mathcal{P}$ is encoded into an element $(p_1, \ldots, p_n) \in \mathrm{B}^\mathrm{n}$, such that:

$$p_i = \begin{cases} 1 & \text{if } p_i \in m \\ 0 & \text{if } p_i \notin m \end{cases} .$$

As an example, both the vertex $(1,0,1,0,0) \in \mathrm{B}^5$ and the cube $p_1\bar{p}_2p_3\bar{p}_4\bar{p}_5$ represent the marking in which p_1 and p_3 are marked whereas p_2, p_4 and p_5 are not marked.



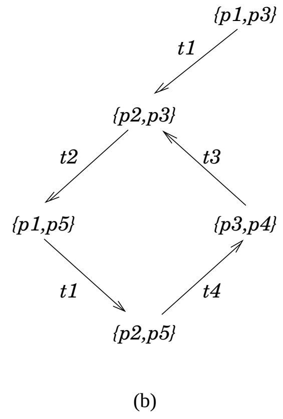


Figure 3.2 (a) A Petri net and (b) its reachable markings.

Given the one-to-one encoding function $\mathcal{E}$, each marking $m \in \mathcal{M}_\mathcal{P}$ can be represented as a minterm of an n-variable logic function, being $n = |\mathcal{P}|$. This minterm is the characteristic function of marking m, and it is written χ_m. The characteristic function of a set of markings M, denoted χ_M, can be obtained as the logic sum of the minterms of each marking $m \in M$. For example, given the Petri net depicted in figure 3.2(a), the characteristic function of the set

$$M = \{\{p_2, p_5\}, \{p_2, p_3, p_5\}, \{p_1, p_2, p_5\}, \{p_1, p_2, p_3, p_5\}, \{p_1, p_2, p_3, p_4, p_5\}\}$$

is calculated as the disjunction of each boolean code $\mathcal{E}(m), m \in M$. The resulting function

$$\chi_M = p_1p_2p_3p_5 + p_2\bar{p}_4p_5$$

represents the set of markings in which p_1, p_2, p_3, and p_5 are marked or p_2 and p_5 are marked and p_4 is not marked.

All set manipulations can by applied directly to the characteristic functions. For example, given the sets of markings $M, M' \subseteq \mathcal{M}_\mathcal{P}$:

$$\begin{aligned}\chi_{M \cup M'} &= \chi_M + \chi_{M'} \ , \\ \chi_{M \cap M'} &= \chi_M \cdot \chi_{M'} \ , \\ \chi_{\overline{M}} &= \overline{\chi_M} \cdot \chi_{\mathcal{M}_\mathcal{P}} \ .\end{aligned}$$

Hence and for sake of simplicity both M and χ_M will be indistinctly used to denote the characteristic function of the set of markings M.

When implemented with BDDs, characteristic functions provide, in general, compact representations and, consequently, operations with sets can be efficiently handled.

Characteristic functions can also be used to represent *binary relations*, that is, subsets of a Cartesian product between two sets. To represent the binary relation $R \subseteq M \times M'$, it is necessary to use different sets of variables to identify the elements of M and M'. Given the binary relation R between sets M and M', the elements of M that are in relation with some element of M', are the set:

$$V = \{m \in M | \exists m' \in M', (m, m') \in R\} \ ,$$

and using the characteristic function of R, the characteristic function of V is computed by:

$$\chi_V(x_1, \ldots, x_n) = \exists_{y_1, \ldots, y_n} \chi_R(x_1, \ldots, x_n, y_1, \ldots, y_n) \ .$$

3.2.2 Firing Petri net transitions

Given a Petri net $\langle \mathcal{P}, \mathcal{T}, \mathcal{F}, m_0 \rangle$, we define the *next state function* as a function

$$\delta_N : 2^{\mathcal{M}_\mathcal{P}} \times \mathcal{T} \to 2^{\mathcal{M}_\mathcal{P}} \ , \tag{3.1}$$

that transforms, for each transition, a set of markings M into a new set of markings M' as follows:

$$\delta_N(M, t) = M' = \{m' \in \mathcal{M}_\mathcal{P} | \exists m \in M, \ m[t\rangle m'\} \ ,$$

i.e. M' is the set of markings reachable from M by firing transition t. Using the terminology for verification of sequential state machines, δ_N performs the *constrained image computation* of the transition t.

Two techniques have been commonly used to calculate next state functions using BDDs: the *transition function* δ_N and the *transition relation* associated to δ_N. We propose a method to implement the transition function δ_N that uses the topology associated to a transition, hence the name *topological image computation*. We refer the reader to [CBM89] and [BCL$^+$94] for the other techniques.

Constrained image computation for transitions can be efficiently implemented by using the topological information of the Petri net and the characteristic function of sets of markings. First, we will present the characteristic function of some important sets related to a transition $t \in \mathcal{T}$:

$$
\begin{aligned}
\mathrm{E}_t &= \prod_{p_i \in {}^\bullet t} p_i && (t \text{ is enabled}),\\
\mathrm{NPM}_t &= \prod_{p_i \in {}^\bullet t} \bar{p}_i && (\text{no predecessor of } t \text{ is marked}),\\
\mathrm{ASM}_t &= \prod_{p_i \in t^\bullet} p_i && (\text{all successors of } t \text{ are marked}),\\
\mathrm{NSM}_t &= \prod_{p_i \in t^\bullet} \bar{p}_i && (\text{no successor of } t \text{ is marked}).
\end{aligned}
$$

Given these characteristic functions, the constrained image computation for transitions is reduced to calculate:

$$\delta_N(M,t) = (M\!\downarrow_{\mathrm{E}_t} \cdot \mathrm{NPM}_t)\!\downarrow_{\mathrm{NSM}_t} \cdot \mathrm{ASM}_t \ . \tag{3.2}$$

We will show with an example how this formula mimics transition firing. In the example of figure 3.2(a), given the set of markings

$$M = p_1\bar{p}_2p_3\bar{p}_4\bar{p}_5 + \bar{p}_1p_2p_3\bar{p}_4\bar{p}_5 + p_1\bar{p}_2\bar{p}_3\bar{p}_4p_5 \ ,$$

we will calculate $M' = \delta_N(M,t_1)$. First, $M\!\downarrow_{\mathrm{E}_{t_1}}$ (the cofactor of M with respect to $\mathrm{E}_{t_1} = p_1$) selects those markings in which t_1 is enabled and removes its predecessor places from the characteristic function:

$$M\!\downarrow_{\mathrm{E}_{t_1}} = \bar{p}_2p_3\bar{p}_4\bar{p}_5 + \bar{p}_2\bar{p}_3\bar{p}_4p_5 \ .$$

Then the product with $\mathrm{NPM}_{t_1} = \bar{p}_1$ simulates the elimination of the tokens in the predecessor places:

$$M\!\downarrow_{\mathrm{E}_{t_1}} \cdot \mathrm{NPM}_{t_1} = \bar{p}_1\bar{p}_2p_3\bar{p}_4\bar{p}_5 + \bar{p}_1\bar{p}_2\bar{p}_3\bar{p}_4p_5 \ .$$

Next, taking the cofactor with respect to $\mathrm{NSM}_{t_1} = \bar{p}_2$ removes all successor places from the characteristic function:

$$(M\!\downarrow_{\mathrm{E}_{t_1}} \cdot \mathrm{NPM}_{t_1})\!\downarrow_{\mathrm{NSM}_{t_1}} = \bar{p}_1p_3\bar{p}_4\bar{p}_5 + \bar{p}_1\bar{p}_3\bar{p}_4p_5 \ .$$

Finally, the product with $\mathrm{ASM}_{t_1} = p_2$ adds a token in all the successor places of t_1:

$$M' = \bar{p}_1p_2p_3\bar{p}_4\bar{p}_5 + \bar{p}_1p_2\bar{p}_3\bar{p}_4p_5 \ .$$

Note that (3.1) is correctly defined only for safe Petri nets. It will be shown in section 3.8 how safeness can be verified.

3.3 SIGNAL TRANSITION GRAPHS (STG)

An interpreted Petri net proposed to specify asynchronous circuit behavior was independently presented by Rosenblum and Yakovlev [RY85], who called it *Signal Graph*, and by Chu [Chu86], who named it *Signal Transition Graph*. Both coincided in interpreting the transitions in the Petri net as the switching of signals. A transition of signal v from 0 to 1 is labeled as $v+$, whereas a 1 to 0 change is labeled $v-$. The symbol $v*$ denotes either $v+$ or $v-$.

Formally, a Signal Transition Graph is a quadruple $\langle \mathcal{N}, \mathcal{V}, \lambda, s^0 \rangle$, where

$\mathcal{N}$ is a Petri net,
$\mathcal{V} = \{v_1, \ldots, v_n\}$ is a set of signal variables,
$\lambda : \mathcal{T} \longrightarrow (\mathcal{V} \times \{+, -\}) \cup \tau$ is a labeling function for transitions and
s^0 is the initial state.

An STG can have transitions that are not associated to any signal, also called *silent transitions*. Such transitions are mapped to the *silent label* τ. STGs are often shown in their shorthand form, where transitions are denoted simply by their labels, without bars nor boxes, and places with only one predecessor and one successor are omitted. The fact of one of such places being marked is denoted by placing the tokens on the arc. Figure 3.3 depicts a Petri net and an equivalent STG.

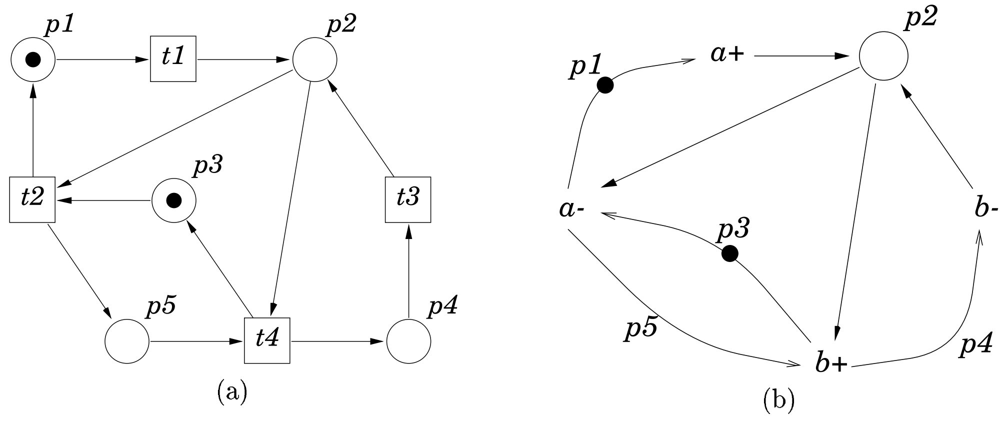


Figure 3.3 Isomorphic (a) Petri net and (b) STG.

The sets of transitions in the Petri net that specify a rising or falling transition of signal v are respectively denoted by T_{v+} and T_{v-}. The symbol T_{v*} is used to denote either T_{v+} or T_{v-}.

For similarity with Petri nets, $s[t\rangle s'$ will denote that state s' is reachable from state s by firing transition t. The set of reachable states of an STG will be written $[s^0\rangle$.

3.3.1 Modeling safe STGs

As Signal Transition Graphs are interpreted Petri nets, the state of an STG is given both by the marking of the underlying Petri net and the values of the signals of the STG. Given an STG $\langle \mathcal{N}, \mathcal{V}, \lambda, s^0 \rangle$, let us call $\mathcal{S}_\mathcal{V}$ the set of all possible combinations of values of STG signals. Considering $\mathcal{S} = \mathcal{M}_\mathcal{P} \times \mathcal{S}_\mathcal{V}$ to be the set of all possible STG states, the system

$$(2^\mathcal{S}, \cup, \cap, \emptyset, \mathcal{S})$$

is the boolean algebra of sets of STG states, which is isomorphic to the boolean algebra of n-variable logic functions, being $n = |\mathcal{P}| + |\mathcal{V}|$ (see Stone's theorem in section 2.4.4).

Consequently, the state of an STG is defined by the ordered pair

$$(p_1, \ldots, p_{|\mathcal{P}|}, v_1, \ldots, v_{|\mathcal{V}|}) \enspace ,$$

where p_i and v_i respectively denote place and signal variables. Similarly as we did with Petri net markings, STG states will be represented by minterms and sets of states by the sum of the minterms of the states in the set.

3.3.2 Firing STG transitions

STG transitions are interpreted as signal transitions, therefore the next state function must simulate the fact that firing a transition also switches the value of the signal. A new image computation function

$$\delta_S : 2^{\mathcal{S}} \times \mathcal{T} \longrightarrow 2^{\mathcal{S}}$$

has to be defined. Since STG transitions are fired following the same rules of Petri nets, δ_S can be computed as follows:

$$\delta_S(S,t) = \begin{cases} \delta_N(S,t){\downarrow_{\bar{v}}} \cdot v \ , & \text{if } t \in T_{v+} \\ \delta_N(S,t){\downarrow_{v}} \cdot \bar{v} \ , & \text{if } t \in T_{v-} \end{cases} .$$

In the sequel, E_{v+} and E_{v-} will denote the set of the states in which signal v is enabled to rise and fall, respectively. The symbol E_{v*} will denote either E_{v+} or E_{v-}. The set E_{v*} can be calculated as

$$\mathrm{E}_{v*} = \bigcup_{t \in T_{v*}} \mathrm{E}_t \ .$$

3.4 STATE GRAPHS

A *State Graph* is a quadruple $\langle S, E, V, \lambda \rangle$, where

S is a set of states,
$E \subseteq S \times S$ is a set of edges (or transitions),
$V = \{v_1, \ldots, v_n\}$ is a set of boolean variables and
$\lambda : S \longrightarrow \{0,1\}^n$ is a total labeling function that encodes each state with a *binary vector* of values.

If the labeling function is the identity, i.e. $\lambda \equiv \mathrm{id}_{\mathrm{B}^n}$, the state graph is denoted $\langle S, E, V \rangle$.

A *path* in the state graph is defined as a sequence of states $s^1, \ldots, s^k$ such that $s^i E s^{i+1}$ for $1 \le i < k$. The fact that path π leads from state s to state s' will be denoted by $s \overset{\pi}{\rightsquigarrow} s'$. We will refer to $|\pi|$ as the number of arcs in a path π.

%

3.4.1 Projection of a state graph

Let us assume $S \subseteq \mathrm{B}^{\mathrm{n}}$, and X as the following subset of variables

$$X = \{v_1, \ldots, v_k\} \subseteq V = \{v_1, \ldots, v_n\}, \text{ with } |X| = k \le n \ .$$

Without loss of generality, we assume X to be the first k elements of V.

The function *projection* for states $s = (s_1, \ldots, s_n) \in S$, $proj : S \longrightarrow \mathrm{B}^{\mathrm{k}}$, is defined as

$$proj_X(s) = (s_1, \ldots, s_k) \ .$$

Definition 3.4 (Projection of a state graph) *Given a state graph, $SG = \langle S, E, V \rangle$, and a subset of variables X defined as above, the projection of SG onto X is a state graph, $proj_X(SG) = \widehat{SG} = \langle \widehat{S}, \widehat{E}, X \rangle$, such that*

$$\begin{aligned} \widehat{S} &= \{\widehat{s} \mid \exists s \in S \text{ with } \widehat{s} = proj_X(s)\} \ , \\ \widehat{E} &= \{(proj_X(s^1), proj_X(s^k)) \mid \exists \pi = s^1, \ldots, s^k \text{ with } proj_X(s^1) = \ldots = proj_X(s^{k-1})\} \ . \end{aligned}$$

Note that the definition of a state as a bit-vector implies that semantically different states can be projected onto the same state, i.e. $proj_X(s) = proj_X(s') = \widehat{s}$ and $s \neq s'$.

3.4.2 State graph associated to a Petri net or an STG

For a Petri net $\langle \mathcal{P}, \mathcal{T}, \mathcal{F}, m_0 \rangle$, we can define its associated state graph $\langle S, E, V \rangle$ such that

$$\begin{aligned} S &= [m_0\rangle, \\ E &= \{(m, m') \in S \times S \mid \exists t \in \mathcal{T} \text{ with } m[t\rangle m'\}, \\ V &= \{p_1, \ldots, p_n\}, n \text{ being } |\mathcal{P}|. \end{aligned}$$

The state graph associated to an STG $\langle \mathcal{N}, \mathcal{V}, \lambda, s^0 \rangle$ is such that

$$\begin{aligned} S &= [s^0\rangle, \\ E &= \{(s, s') \in S \times S \mid \exists t \in \mathcal{T} \text{ with } s[t\rangle s'\}, \\ V &= \{p_1, \ldots, p_n, v_1, \ldots, v_m\}, \text{ with } n = |\mathcal{P}|, m = |\mathcal{V}|. \end{aligned}$$

3.5 NET TRAVERSAL AND REACHABLE MARKINGS

This section presents how the set of reachable markings of a Petri can be efficiently computed by using symbolic techniques. The extension of all those concepts to Signal Transition Graphs is straightforward.

Given a Petri net $\langle \mathcal{P}, \mathcal{T}, \mathcal{F}, m_0 \rangle$ the constrained image computation function of the net is a function

$$\Delta_N : 2^{\mathcal{M}_\mathcal{P}} \longrightarrow 2^{\mathcal{M}_\mathcal{P}}$$

that given a set of markings M, computes the set of markings reachable from M by firing one transition. The function Δ_N can be defined as follows:

$$\Delta_N(M) = \{m' \in \mathcal{M}_\mathcal{P} \mid \exists m \in M, t \in \mathcal{T} \text{ such that } m[t\rangle m'\} \ .$$

Let us call the set M' the image of M under Δ_N, i.e. $M' = \Delta_N(M)$. The set M' can be alternatively computed by using the next state functions for transitions, δ_N, as follows:

$$M' = \bigcup_{t \in \mathcal{T}} \delta_N(M, t) \ .$$

Let us call S^i the set of markings reachable from the initial marking m_0 by firing at most i transitions. The different S^i sets can be formally defined by the following recursion:

1. $S^0 = \{m_0\}$.
2. $S^i = \Delta_N(S^{i-1})$.

We can also define the ith frontier, denoted F^i, as the markings in S^i that are not in S^{i-1}, i.e.

$$F^i = S^i - S^{i-1} \ .$$

Clearly, the set of reachable markings $[m_0\rangle$ can be calculated as

$$[m_0\rangle = \bigcup_{i \geq 0} S^i = (\bigcup_{i \geq 1} F^i) \cup \{m_0\} \ .$$

Note that each S^i is a superset of the previous ones. Since for bounded Petri nets the number of reachable markings is finite, there is some k such that $S^k = S^{k+1}$. The problem of computing the set of reachable markings of a bounded Petri net can be reduced to finding a *fixed point* of function Δ_N, i.e. some S^k such that

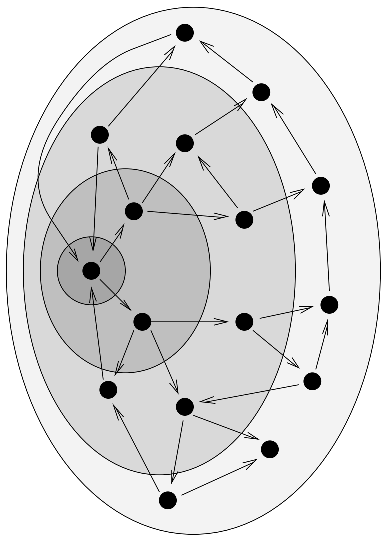
$$S^k = \Delta_N(S^k) \ .$$

The S^k set with smallest k such that the previous formula holds is called the *least fixed point*. Recall that the least fixed point is equal to the reachable set of markings $[m_0\rangle$.

The least fixed point of function Δ_N can be calculated by *symbolic traversal* of the Petri net. We will use an approach similar to the *symbolic Breadth First Search* (BFS) algorithm for traversing Finite State Machines [CBM89]. This method allows to process several markings simultaneously by using their characteristic function and the constrained image computation. The algorithm presented in figure 3.4(a) traverses the Petri net and calculates $[m_0\rangle$. The union and difference of sets are performed by manipulating their characteristic functions.

```
Breath_First_Search ($N = \langle \mathcal{P}, \mathcal{T}, \mathcal{F}, m_0 \rangle$) {
   Reached := From := $m_0$;
   do {
      New := $\emptyset$;
      for each transition $t \in \mathcal{T}$ {
         New := New $\cup$ $\delta_N$(From,$t$);
      }
      From := New - Reached;
      Reached := New $\cup$ Reached;
   } while (From $\neq \emptyset$);
}
```

(a)



(b)

Figure 3.4 (a) Algorithm for symbolic traversal and (b) graphical representation of Breadth First Search.

During the ith iteration of the do loop the variable From contains the frontier set F^{i-1}. The innermost loop computes all the markings that are reachable from F^{i-1} by firing one transition and accumulates them in the variable New. At the end of the ith iteration of the do loop, the variable From is assigned the markings in New that are really new, i.e. the ith frontier F^i. Consequently, each iteration of the do loop computes a new S^i set, which is kept in the variable Reached. The BFS algorithm finishes when no new markings are generated. This situation is produced when a least fixed point is reached, i.e. when $S^i = S^{i-1}$ or equivalently when $F^i = \emptyset$. The number of iterations performed by the algorithm is determined by the maximum number of firings from the initial marking to the first occurrence of any of the reachable markings, and it is called the *sequential depth* of the Petri net. Figure 3.4(b) illustrates graphically how the set of reachable markings is computed.

3.6 PETRI NET REDUCTIONS

Petri nets can be reduced to simpler ones by using transformation rules that preserve the properties of the system being modeled. By using these rules, the complexity inherent to the reachability analysis can be effectively reduced.

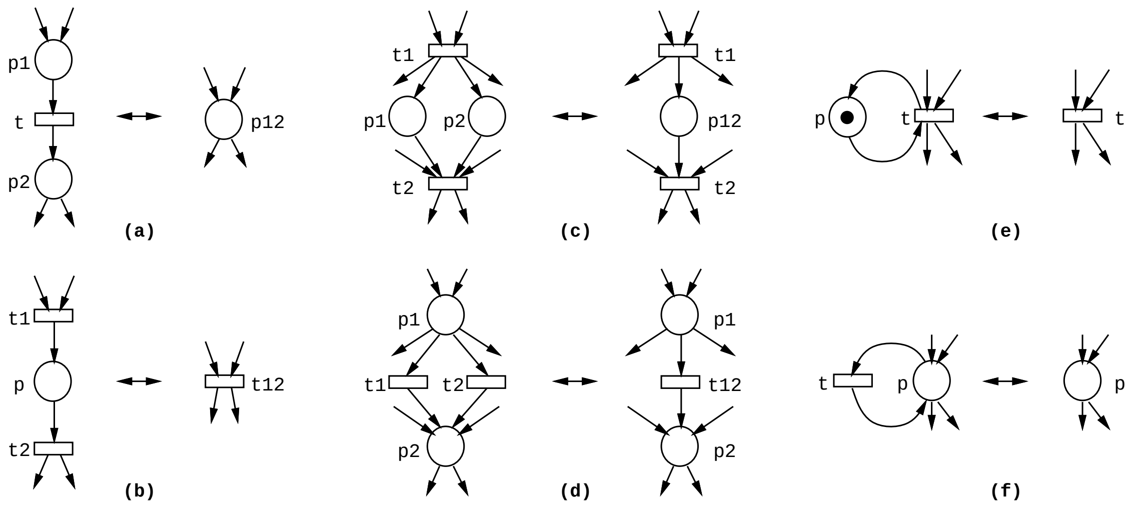


Figure 3.5 Transformations preserving liveness, safeness and boundedness.

Murata [Mur89] proposed a set of six transformations that preserve the properties of liveness, safeness, and boundedness. The transformations we use are slightly different from those in [Mur89], since they not only preserve these properties, but also allow recovering the original set of markings. Figure 3.5 depicts the set of transformations actually used.

The original Petri net N is reduced to a new net N' by applying these transformations. Then, the reachability analysis technique presented in section 3.5 can be used more efficiently with N' due to the reduction in both, the number of places and the sequential depth of the net. Given the set of reachable markings $[m'_0\rangle$ of N', the set of reachable markings $[m_0\rangle$ of the original net N is derived using an inverted transformation on $[m'_0\rangle$. The inverted transformations are shown in table 3.1.

For example, figure 3.5(a) depicts how a net can be transformed into another by fusing places p_1 and p_2 into place p_{12}. If R' is the set of reachable markings of the resulting net, the set of markings in the original net, R, can be derived as follows:

$$R = R' \downarrow_{p_{12}} \cdot (p_1 \oplus p_2) + R' \downarrow_{\bar{p}_{12}} \cdot \bar{p}_1 \cdot \bar{p}_2 \ ,$$

Forward Transformations	Backward Transformations
(a) series places fusion	$R = R'\downarrow_{p_{12}} \cdot (p_1 \oplus p_2) + R'\downarrow_{\bar{p}_{12}} \cdot \bar{p}_1 \cdot \bar{p}_2$
(b) series transitions fusion	$R = R'\downarrow_{\mathrm{E}_{t_{12}}} \cdot (\mathrm{E}_{t_1} \cdot \bar{p} + \mathrm{NPM}_{t_1} \cdot p) + R'\downarrow_{\overline{\mathrm{E}}_{t_{12}}} \cdot \overline{\mathrm{E}}_{t_1} \cdot \overline{\mathrm{E}}_{t_2}$
(c) parallel places fusion	$R = R'\downarrow_{p_{12}} \cdot p_1 \cdot p_2 + R'\downarrow_{\bar{p}_{12}} \cdot \bar{p}_1 \cdot \bar{p}_2$
(d) parallel transitions fusion	$R = R'$
(e) self-loop place	$R = R' \cdot p$
(f) self-loop transition	$R = R'$

Table 3.1 Petri net reductions and their inverse transformations.

denoting that a token in p_{12} implies that either p_1 or (exclusive or) p_2 were marked and no token in p_{12} implies that neither p_1 nor p_2 were marked in the original net. Similar substitutions can be applied for other types of transformations.

3.7 IMPROVING PETRI NET TRAVERSAL

Different improvements can be made to the basic Breadth First Search (BFS) algorithm presented in section 3.5. Two different techniques will be presented in order to reduce the number of BDD operations. The first one reduces the number of traversal iterations, while the second one allows to perform image computation by using a new ad hoc BDD operation.

3.7.1 Chaining

This technique drastically reduces the number of traversal iterations. As a direct consequence, the execution time can be reduced between one and three orders of magnitude, depending on the example.

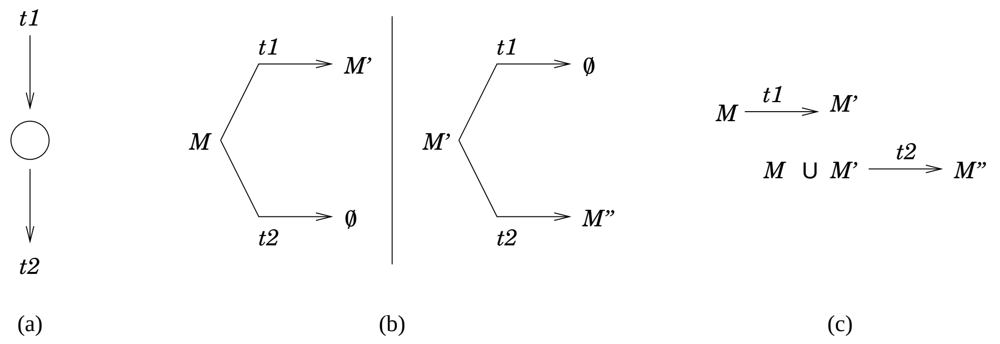


Figure 3.6 (a) Two consecutive transitions. Traversal (b) without or (c) with chaining.

We will introduce the *chaining* technique by means of the example in figure 3.6. Let us assume figure 3.6(a) to be a piece of a safe Petri net. Transitions t_1 and t_2 are consecutive, i.e. t_2 can only fire after t_1 has fired. Given a set of markings $M \subseteq [m_0\rangle$, we will calculate the set of markings reached by firing both transitions. For simplicity we will assume that only t_1 is enabled in M.

Formally,

$$M \cap \mathrm{E}_{t_1} \neq M \cap \mathrm{E}_{t_2} = \emptyset \ .$$

Figure 3.6(b) shows how the first iteration of a BFS algorithm takes `From` $= M$ and computes

$$\begin{aligned} \delta(M, t_1) &= M' \ \text{ and} \\ \delta(M, t_2) &= \emptyset \ . \end{aligned}$$

The second iteration assigns `From` $= M' \cup \emptyset = M'$ and calculates

$$\begin{aligned} \delta(M', t_1) &= \emptyset \ \text{ and} \\ \delta(M', t_2) &= M'' \ . \end{aligned}$$

As we have seen, two iterations are needed to obtain $M' \cup M''$. In addition, time is wasted in the first and second iterations trying unsuccessfully to fire t_2 and t_1 respectively.

The chaining technique tries to avoid this by adding the new markings to the `From` set as soon as they are calculated. In this case, assuming that the initial origin set of markings was M, the algorithm first computes

$$M' = \delta_N(M, t_1) \ ,$$

and then

$$M'' = \delta_N(M \cup M', t_2) \ .$$

As figure 3.6(c) illustrates, the markings reached by firing t_1 and then t_2 are computed in the same traversal iteration.

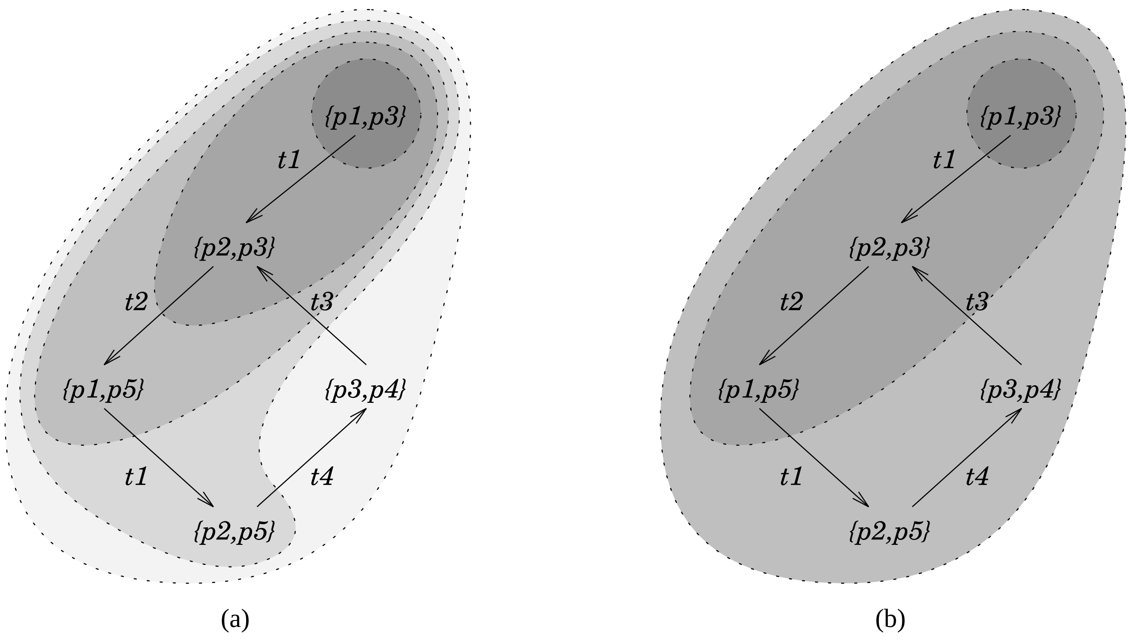


Figure 3.7 (a)Traversal without chaining and (b) chaining.

We can use the Petri net in figure 3.2(a) as an example. Figures 3.7(a) and 3.7(b) show, respectively, the set of markings reached at each iteration when a basic BFS algorithm and chaining technique are used. Without chaining, a BFS algorithm needs as many iterations as the sequential depth of the net, which in this case is four, to reach a fixed point. However, the number of traversal iterations can be reduced if we use this chaining technique. Let us assume that we try to fire transitions in the order t_1, t_2, t_4, t_3. In the first iteration transition t_1 fires, reaching $\{p_2, p_3\}$, and then t_2, reaching $\{p_1, p_5\}$. In the second iteration, transitions t_1, t_4 and t_3 successively fires, and all the reachable markings are computed. This description is illustrated in figure 3.7(b).

Note that different orders may result in different number of iterations. Several algorithms have been tried to order transition firing. We have found that a *token-flow* algorithm through the Petri net structure has turned out as the best heuristic. The complexity of this search is lineal in the size of the net. Other heuristics, e.g. *depth first search* or *breadth first search*, also presented good, and in some cases very similar, results. As a general consideration, chaining, even if transitions are randomly ordered, drastically reduces the execution time of a symbolic traversal algorithm, depending on the example up to tree orders of magnitude. Choosing between one heuristic or another results in comparatively small variations, of at most 50%.

3.7.2 Toggling

Let us review the image computation functions for both Petri nets and STG transitions:

$$\begin{aligned} \delta_N(M,t) &= (M{\downarrow_{\mathrm{E}_t}} \cdot \mathrm{NPM}_t){\downarrow_{\mathrm{NSM}_t}} \cdot \mathrm{ASM}_t \ , \\ \delta_S(S,t) &= \begin{cases} \delta_N(S,t){\downarrow_{\bar{v}}} \cdot v \ , & \text{if } t \in T_{v+} \\ \delta_N(S,t){\downarrow_{v}} \cdot \bar{v} \ , & \text{if } t \in T_{v-} \end{cases} \ . \end{aligned}$$

We could think of those operations as divided into two different parts: the first one devoted to calculating the set of states in which transition t is enabled, and the second one devoted to complement or *toggle* the variables of predecessor and successor places and, in the case of STGs, the value of the signal variable associated to t.

Given a BDD representing a logic function, the fact of complementing a list of variables is equivalent to interchanging the *then* and *else* arcs in each node labeled with one of the variables to complement. In addition, if we represent the list of variables as a cube, we could implement a `BDD_AND_toggle` operation as a function that given two BDDs, F and G, and a set of variables C, performs the product $F \cdot G$ and complements the variables in C. The set C can be represented as a cube containing the literals to be complemented.

We propose the algorithm in figure 3.8 to efficiently compute this operation. The complexity of this algorithm is exponential in the number of BDD nodes, as occurred in the `ite` algorithm described in section 2.6.3. This complexity can be overcome in a similar way by keeping the result of each call in a table of computed results. The function `BDD_AND_toggle` has been integrated in the BDD package by David Long [Lon93].

Therefore, the operation $M' = \delta_N(M,t)$ will be equivalent to executing

$$M' \ := \texttt{BDD\_AND\_toggle}\big(M, \mathrm{E}_t, (\mathrm{E}_t \cup \mathrm{ASM}_t) - (\mathrm{E}_t \cap \mathrm{ASM}_t)\big) \ .$$

For STGs, $M' = \delta_S(M,t)$, assuming $t \in T_{v*}$, is computed as

$$M' \ := \texttt{BDD\_AND\_toggle}\big(M, \mathrm{E}_t, v \cap [(\mathrm{E}_t \cup \mathrm{ASM}_t) - (\mathrm{E}_t \cap \mathrm{ASM}_t)]\big) \ .$$

3.7.3 Improved BFS algorithm

Figure 3.9 shows an improved BFS algorithm that includes the above modifications plus the reduction of the original Petri net. First the net is reduced by applying the transformations presented in section 3.6. Then transitions are ordered following some heuristic. The innermost loop

```
BDD_AND_toggle(F,G,C) {
   if ( terminal case )
      return result for terminal case;
   if ( {F, G, C} is in computed table )
      return pre-computed result;
   let $v$ be the top variable of { F, G, C };
   let $u$ be the top variable of { F, G };
   let $c$ be the top variable of { C };
   if ($c = v \wedge c \neq u$) {
      return BDD_AND_toggle(F,G,$C\!\downarrow_c$);
   T = BDD_AND_toggle($F\!\downarrow_v$,$G\!\downarrow_v$,$C\!\downarrow_v$);
   E = BDD_AND_toggle($F\!\downarrow_{\bar{v}}$,$G\!\downarrow_{\bar{v}}$,$C\!\downarrow_v$);
   if ($c \neq u$)
      R := find_or_add_unique_table($v$,T,E);
   else
      R := find_or_add_unique_table($v$,E,T);
   insert_computed_table({ F, G, C }, R);
   return R;
}
```

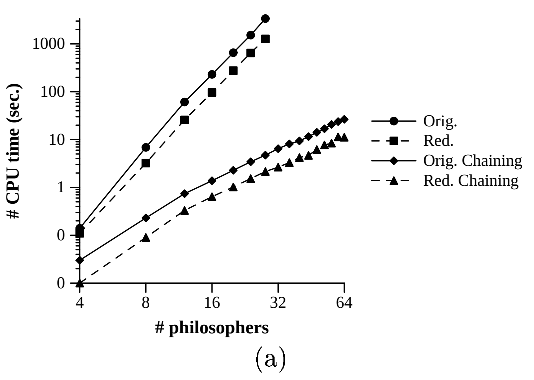
Figure 3.8 BDD AND-toggle algorithm.

```
improved_BFS ($N = \langle \mathcal{P}, \mathcal{T}, \mathcal{F}, m_0 \rangle$) {
   $N'$ = reduce($N$);
   $\mathcal{T}'_{\mathrm{Ord}}$ = order_transitions($N'$);
   Reached := From := $m'_0$;
   do {
      From := $\emptyset$;
      for each transition $t \in \mathcal{T}'_{\mathrm{Ord}}$ {
         From := From $\cup$ $\delta$(From,$t$);
      }
      From := From - Reached;
      Reached := From $\cup$ Reached;
   } while (From $\neq \emptyset$);
   Reached := expand_markings($N$,$N'$,Reached);
}
```

Figure 3.9 Improved BFS algorithm.

implements chaining technique, while the other instructions in the do loop are the same as in the basic BFS algorithm in section 3.5. Finally, the set of reachable markings of the reduced Petri net is expanded to the reachable markings of the original net by applying the inverse transformations described in section 3.6.

We show the results of the techniques above applied to a Petri net describing the *dining philosophers* paradigm. The benchmark is further discussed in section 3.9. Figure 3.10(a) illustrates how the chaining technique reduces the execution time in more than three orders of magnitude as the size of the Petri net increases. This spectacular improvement is due to the fact that, in this example, an appropriate transition firing order keeps the number of traversal iterations constant, regardless the size of the Petri net. Figure 3.10(b) shows that by using the ad-hoc toggling operation the CPU time is reduced to the half. The reason for this gain comes from the reduction in the number of BDD operations when computing the image computation, δ. As both figures depict, the improvement obtained by chaining and toggling is orthogonal to the use of reduction techniques.



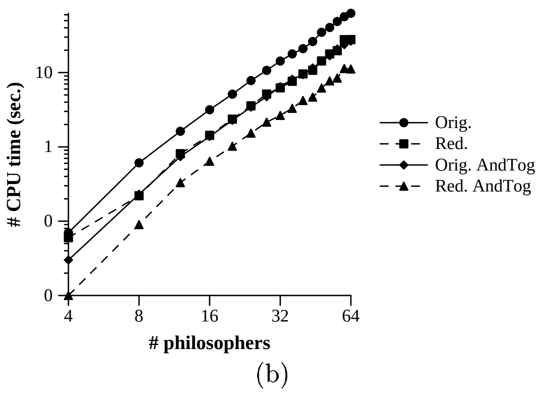


Figure 3.10 Improvements obtained by (a) chaining and (b) toggling techniques on the dining philosophers example.

Yoneda et al. [YHTM96] have also proposed an ad hoc operation for image computation. Surprisingly, in some examples their results are just the opposite of what should be expected, since the ad hoc operation is slower than image computation calculated by using explicit BDD operations. As pointed out by the authors, that strange behavior might appear because they used different BDD packages in each experiment.

3.8 VERIFICATION OF PROPERTIES

Different synthesis methodologies use Signal Transition Graphs as the specification of the circuits to be synthesized [RY85, Chu87]. Specification validation is a previous step to the synthesis process, which must guarantee that the specification satisfies some properties. Kondratyev et al. proved in [KCK+95] that a Signal Transition Graph is implementable as a speed-independent circuit if it satisfies the properties of boundedness, signal persistence, consistence and complete state coding. Other properties like different levels of liveness are simply desirable. In the rest of this section these properties will be described and algorithms to prove them will be presented.

In this section we show how several Petri net and STG properties can be verified by boolean manipulation on the set of reachable states. Some properties can be easily specified with a boolean equation, thus not requiring any traversal to be verified. Others require partial or complete traversals of the net. However, symbolic traversal by means of BDDs makes their computation affordable even for large nets.

All algorithms assume that either a Petri net

$$N = \langle \mathcal{P}, \mathcal{T}, \mathcal{F}, m_0 \rangle$$

or a Signal Transition Graph

$$D = \left\langle \langle \mathcal{P}, \mathcal{T}, \mathcal{F}, m_0 \rangle, \mathcal{V}, \lambda, s^0 \right\rangle$$

have been traversed and that the reachable set of markings $[m_0\rangle$ or states $[s^0\rangle$ has been computed.

3.8.1 Safeness

The calculation of $[m_0\rangle$ by means of constrained image computation is done under the assumption that the Petri net is safe. This calculation is erroneous if some of the markings is unsafe[1], since unsafe markings are not representable by encoding each place with one boolean variable. A similar reasoning can be done for k-bounded nets [PRC97].

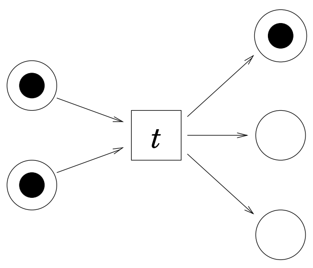
The operations performed in equation (3.2) are correct only for safe Petri nets. However, detecting if some unsafe marking is reachable from $[m_0\rangle$ can be done by identifying a marking m in which a transition t is enabled, and some successor place p of t, and not predecessor of t, is already marked (see figure 3.11(a)). Such marking will be referred as *pre-unsafe* markings. In that situation, after firing transition t, place p will have two tokens. Formally:

$$\begin{aligned} N \text{ is not safe} \quad \Leftrightarrow \quad & \exists (m \in [m_0\rangle, t \in \mathcal{T}, p \in \mathcal{P}) \text{ such that} \\ & t \text{ is enabled in } m,\ p \in t^\bullet,\ p \notin {}^\bullet t \text{ and } m(p) = 1. \end{aligned}$$

We can define the characteristic function of pre-unsafe markings associated to transition t as follows:

$$\texttt{PreUnsafe} = \mathrm{E}_t \cap \left(\bigcup_{p_i \in (t^\bullet - {}^\bullet t)} p_i \right) .$$

The algorithm in figure 3.11(b) calculates the characteristic function of pre-unsafe markings associated to t, and then computes the intersection with the reachable set of markings. If that intersection is not empty, the net is not safe.



(a)

```
is_safe($N$,Reached) {
    foreach transition $t \in \mathcal{T}$ do {
        Succ_marked := $\emptyset$;
        for each place $p_i \in (t^\bullet - {}^\bullet t)$ do {
            Succ_marked := Succ_marked $\cup$ $p_i$;
        }
        PreUnsafe := $\mathrm{E}_t \cap$ Succ_marked;
        if (PreUnsafe $\cap$ Reached $\neq \emptyset$)
            return FALSE;
    }
    return TRUE;
}
```

(b)

Figure 3.11 (a) Firing t produces an unsafe marking. (b) Algorithm for safeness checking.

3.8.2 Liveness

A Petri net is said to have a *deadlock* if there is a marking in which no transition can be fired. A transition is said to be *dead* (L0-live following [Mur89]) if it is never enabled in $[m_0\rangle$. A transition that can be fired at least once in some firing sequence from m_0 is said to be *potentially fireable* (L1-live [Mur89]). All these properties can be verified with simple equations.

The set of markings where a *deadlock* occurs is calculated:

$$\texttt{Deadlock} = [m_0\rangle \cap (\bigcup_{t \in \mathcal{T}} \overline{\mathrm{E}_t}) .$$

[1] In this context, unsafe markings are those with more than one token is some place.

The set of markings where a transition is *potentially fireable* is calculated as:

$$\texttt{Fireable}_t = [m_0\rangle \cap \mathrm{E}_t \ .$$

If $\mathrm{Fireable}_t = \emptyset$, then transition t is L0-live, otherwise it is at least L1-live.

Verifying whether a transition can be fired an infinite number of times from some or any marking of $[m_0\rangle$, i.e. if a transition is L3-live or L4-live [Mur89], requires more elaborate techniques. Both problems can be reduced to the calculation of the *Strongly Connected Components* of $[m_0\rangle$.

Definition 3.5 (SCC) *A Strongly Connected Component* (SCC) *U of a directed graph $G = (V, E)$, is a maximal set of vertices $U \subseteq V$, such that for every pair of vertices u and v in U we have both $u \rightsquigarrow v$ and $v \rightsquigarrow u$; that is, vertices u and v are reachable from each other.*

Definition 3.6 (TSCC) *A Strongly Connected Component U of a directed graph $G = (V, E)$ is* terminal (TSCC) *if from the vertices in U it is not possible to reach any vertex in $V - U$.*

A transition t enabled in all the TSCCs markings of the Petri net is L4-live, because from any marking of $[m_0\rangle$ we will reach some TSCC_i where t can be fired an infinite number of times. L4-liveness of transition t can be computed as follows:

$$t \text{ is L4-live} \iff \bigwedge_{\forall i} (\mathrm{TSCC}_i \cap \mathrm{E}_t \neq \emptyset) \ .$$

If there is some SCC_i where transition t is enabled, then t is L3-live because there is at least a firing sequence from $[m_0\rangle$ that leads to TSCC_i where t can be fired an infinite number of times. L3-liveness for transition t can be calculated as follows:

$$t \text{ is L3-live} \iff \bigvee_{\forall i} (\mathrm{SCC}_i \cap \mathrm{E}_t \neq \emptyset) \ .$$

The algorithm to compute the TSCCs and SCCs of $[m_0\rangle$ is shown in figure 3.12. First, the *transitive closure* (T_R^+) of the *transition relation* is computed, where $T_R^+(x, y) = 1$ if there is a firing sequence from x that leads to y ($x \rightsquigarrow y$) [Bur90]. The following steps compute the sets of markings that are in any SCC (InSCC) or in any TSCC (InTSCC). Finally, each individual SCC (TSCC) is obtained from InSCC (InTSCC).

3.8.3 Persistence

A Petri net is said to be persistent if, for any two enabled transitions, the firing of one transition does not disable the other.

The algorithm in figure 3.13(a) verifies persistence for a Petri net. For each transition t_1, the set of markings with t_1 enabled are calculated. Next, the sets of markings reachable in one step by firing any transition different from t_1 are obtained. If t_1 is not enabled in some of those markings, then the net is not persistent.

In terms of an STG, the idea of transition persistence is not as important as the concept of *signal persistence*. Signal persistence means that an enabled signal will fire independently from the firing of other signals.

```
/* Let $T_R$ be the Transition Relation of $N$ */
compute_SCC_TSCC ($N = \langle P, T, F, m_0 \rangle$, $[m_0\rangle$) {
   $T_R^+$ := compute_Transitive_Closure($T_R$);
   $C_Y$ := $T_R^+(x,y) \cdot T_R^+(y,x)$;
   $C_{NY}$ := $T_R^+(x,y) \cdot \overline{T_R^+(y,x)}$;
   InSCC := $\exists_y C_Y(x,y)$;
   InTSCC := $(\exists_y C_{NY}(x,y))'$;
   SCC$_{1 \ldots m}$ := extract_Strongly_Connected_Components(InSCC);
   TSCC$_{1 \ldots m}$ := extract_Strongly_Connected_Components(InTSCC);
}
```

Figure 3.12 Algorithm to compute the SCC and TSCC sets of $[m_0\rangle$.

```
is_t_persistent ($\langle \mathcal{P}, \mathcal{T}, \mathcal{F}, m_0 \rangle$) {
   foreach $p \in \mathcal{P}, |p^\bullet| > 1$ do {
      foreach $t_i \in p^\bullet$ do {
         Enabled := Reached $\cap$ $\mathrm{E}_{t_i}$;
         foreach $t_j \in p^\bullet, t_j \neq t_i$ do {
            if ($\delta$(Enabled, $t_j$) $\cap$ $\overline{\mathrm{E}_{t_i}} \neq \emptyset$)
               return FALSE;
         }
      }
   }
   return TRUE;
}
```

(a)

```
is_s_persistent ($\langle \mathcal{P}, \mathcal{T}, \mathcal{F}, m_0 \rangle, \mathcal{V}, \lambda$) {
   foreach $p \in \mathcal{P}, |p^\bullet| > 1$ do {
      foreach $t_i \in p^\bullet$ do {
         Enabled := Reached $\cap$ $\mathrm{E}_{t_i}$;
         foreach $t_j \in p^\bullet, t_j \neq t_i$ do {
   /* Let $\lambda(t_i) = a{*}/i$ and $\lambda(t_j) = b{*}/j$ */
            if ($\delta$(Enabled, $t_j$) $\cap$ $\overline{\mathrm{E}_{a*}} \neq \emptyset$)
               return FALSE;
         }
      }
   }
   return TRUE;
}
```

(b)

Figure 3.13 Algorithms to verify (a) transition persistence and (b) signal persistence.

However, we should distinguish between input and non-input signals. As input signals are controlled by the environment, it is possible to have non-deterministic choice between several input signals, i.e. firing an input signal disables other enabled input signal transitions. Such conflicts are always interpreted as choice and therefore do not lead to hazardous behavior. For non-input signals, driven by circuit gates should the circuit be implemented, signal transition disabling my produce a *glitch* or *voltage spike* that may well cause circuit malfunction. Consequently, non-persistence must be avoided when it involves some non-input signal. The algorithm in figure 3.13(b) illustrates how signal persistence can be verified.

Transition and signal persistence are closely related. Clearly, the only source of non-persistence of signal v is the non-persistence of some transition labeled $v*$. Therefore transition persistence implies signal persistence, but not the opposite. The example in figure 3.14 shows the subtle difference between transition and signal persistence. Figure 3.14(a) presents non-persistent transitions, as firing transition $a+$ disables $b+$ and vice versa. Depending on the fired transition, the forthcoming events will be different, either $c-$ or $d+$. On the contrary, the STG in figure 3.14(b) is transition non-persistent but signal persistent. Numbers following transitions are used to distinguish different transitions of a same signal. Transitions $a+/1$ and $b+/1$ are non-persistent, since there is an evident choice between them. However, the STG is signal persistent, as firing transition $a+/1$ disables $b+/1$ but immediately enables $b+/2$, hence signal b is always enabled. The case with the other branch of the choice is symmetric. Finally, figure 3.14(c) depicts a both transition and signal persistent STG with a state graph equivalent to the one of figure 3.14(b).

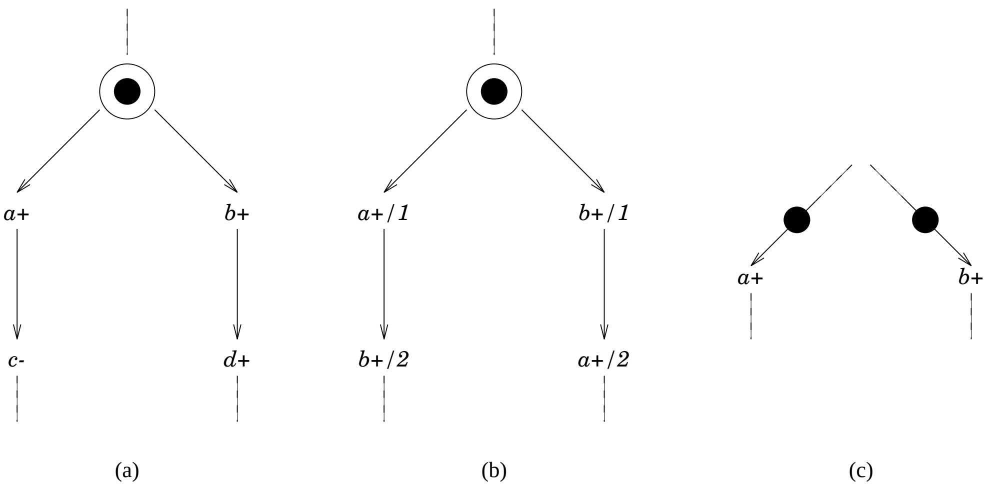


Figure 3.14 Transition and signal (non-)persistence.

3.8.4 Consistence

Not every STG can be associated with a process of switching circuit signals. For example, let us assume that a sequence like $a+/1, b+, a+/2, \ldots$ is feasible in some STG. After firing $a+/1$ the value of signal a is 1 and will not change until an $a-$ transition occurs. Obviously, transition $a+/2$ cannot take place, as a signal cannot switch twice from 0 to 1 without having become 0 in between.

Formally, an STG is consistent iff for each pair of states such that $s[t\rangle s'$ the following conditions are met:

1. If $t \in T_{v+}$, then $v = 0$ in state s and $v = 1$ in state s'.
2. If $t \in T_{v-}$, then $v = 1$ in state s and $v = 0$ in state s'.
3. Any signal different from v has the same value in both s and s'.

Consistence can be checked by an algorithm like the one depicted in figure 3.15.

```
is_consistent($N$,Reached) {
   foreach transition $t \in \mathcal{T}$ do {
      if $t \in T_{v+}$ then
         Inconsistent := Reached $\cap$ $v$;
      else
         Inconsistent := Reached $\cap$ $\bar{v}$;
      if (Inconsistent $\neq \emptyset$)
         return FALSE;
   }
   return TRUE;
}
```

Figure 3.15 Algorithm for consistence checking.

3.8.5 Complete State Coding

Complete State Coding (CSC) is a necessary condition for an STG to be implementable by an asynchronous circuit. A state graph satisfies the CSC requirement iff

1. each state has a unique binary code or
2. for pairs of states that have identical binary codes, the set of non-input enabled signals is the same.

In other words, from any pair of states with the same code, the behavior of the circuit is the same.

For an STG with consistent state assignment, the CSC requirement is violated when two different states with the same binary code produce different values on some non-input signal. CSC can be checked for each non-input signal by defining the following characteristic functions:

$$\begin{aligned}
\mathrm{ER}(v+) &= \mathit{proj}_{\mathcal{V}}(R \cap \mathrm{E}_{v+}) \\
\mathrm{ER}(v-) &= \mathit{proj}_{\mathcal{V}}(R \cap \mathrm{E}_{v-}) \\
\mathrm{QR}(v+) &= \mathit{proj}_{\mathcal{V}}(R \cap v - \mathrm{E}_{v-}) \\
\mathrm{QR}(v-) &= \mathit{proj}_{\mathcal{V}}(R \cap \bar{v} - \mathrm{E}_{v+})
\end{aligned}$$

where R represents the characteristic function of the set of reachable STG states.

$\mathrm{ER}(v*)$ is the set of binary codes that correspond to states in which some transition in T_{v*} is enabled (*a set of excitation regions*). It is obtained by abstracting the places ($\exists_{\mathcal{P}}$) from the states of the excitation region. $\mathrm{QR}(v+)$ (*a set of quiescent regions*) is the set of binary codes that correspond to states in which $v = 1$ but $v-$ is not enabled (similarly for $\mathrm{QR}(v-)$).

The set of states violating the CSC requirement for non-input signal v can now be calculated as follows [PC93]:

$$\mathrm{nonCSC}(v) = \big(\mathrm{ER}(v+) \cap \mathrm{QR}(v-)\big) \cup \big(\mathrm{ER}(v-) \cap \mathrm{QR}(v+)\big) \ .$$

An STG has CSC if all its non-input signals ($\mathcal{V} - \mathcal{V}_I$) satisfy the CSC requirement, i.e. if the following formula holds:

$$\left(\bigcup_{v \in \mathcal{V} - \mathcal{V}_I} \mathrm{nonCSC}(v) \right) = \emptyset \ .$$

3.9 RESULTS

In this section we illustrate the power of using boolean reasoning and BDDs for the analysis of Petri nets and Signal Transition Graphs.

3.9.1 Petri nets

We have chosen two simple and scalable examples to show how the approach can generate all the states for fairly large nets. We present the results corresponding to the calculation of the set of reachable markings, which dominates the complexity of the analysis. Most properties can then be verified in a straightforward manner from $[m_0\rangle$, as shown in section 3.8.

The first example is the well-known *dining philosophers* paradigm, represented by the Petri net in figure 3.16. The net has $7n$ places and $5n$ transitions, n being the number of philosophers sitting at the table. By successively applying the reductions depicted in figure 3.5, the complexity of the net can be reduced down to $6n$ places and $4n$ transitions.

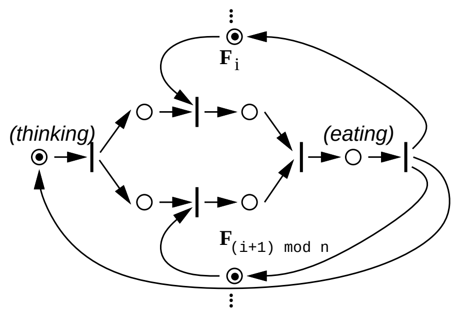


Figure 3.16 Petri net for a dining philosopher.

Table 3.2 shows the number of states of the original and the reduced Petri net, the size of the BDDs representing the reachable markings and the number of iterations and CPU time spent by the traversal algorithm. CPU times have been obtained by executing the algorithms on a Sun SPARCStation-20, with 128 MB memory.

It is worthwhile to point out how a small BDD (2531 nodes ≈ 40 KB memory) can represent the complete set of markings of the Petri net for 64 philosophers (4.9×10^{42}). The BDD representing $[m_0\rangle$ has been calculated by using the traversal algorithm presented in figure 3.9. The algorithm computes $[m_0\rangle$ in only one iteration, although a second iteration is needed to find out that no new markings are generated. This is a clear example of how an appropriate transition firing order can reduce the number of iterations required for the computation of $[m_0\rangle$.

# of	states		BDD size			# of	CPU
philos.	original	reduced	orig.	red.	peak (red.)	iters.	(secs.)
4	466	322	131	106	123	2	0.01
8	2.2×10^5	1.0×10^5	291	234	275	2	0.09
16	4.7×10^{10}	1.1×10^{10}	611	718	579	2	0.64
32	2.2×10^{21}	1.2×10^{20}	1251	1002	1187	2	2.65
64	4.9×10^{42}	1.3×10^{40}	2531	2026	2403	2	11.16

Table 3.2 Results for the *dining philosophers* example.

The second example models a protocol for Local Area Networks called *slotted ring*. The Petri net is depicted in figure 3.17. The example is scalable for any number of nodes in the network. The results corresponding to the traversal of the net are presented in table 3.3.

Figure 3.18 illustrates how the symbolic traversal algorithm in figure 3.9 is capable of manipulating a large set of markings with comparatively small BDD variables. Figure 3.18(a) depicts the number of states represented by the BDD variables `From` and `Reached` at each iteration of the innermost loop. The evolution of the size of those variables during the traversal is shown in figure 3.18(b). The example used for both graphics is the reduced net that specifies the slotted ring protocol with eight nodes.

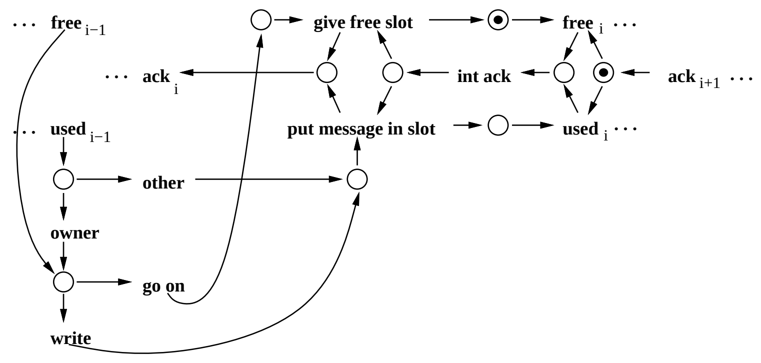


Figure 3.17 Slotted ring protocol for one node.

# of nodes	states original	states reduced	BDD size orig.	BDD size red.	BDD size peak (red.)	# of iters.	CPU (secs.)
4	8.2×10^4	5.1×10^3	443	200	594	5	0.34
6	3.7×10^7	5.8×10^5	950	414	1985	6	3.08
8	1.7×10^{10}	6.8×10^7	1641	704	5804	9	16.9
10	8.5×10^{12}	8.3×10^9	2516	1070	13427	10	70.4
12	4.2×10^{15}	1.0×10^{12}	3575	1512	32559	12	258
14	2.1×10^{18}	1.3×10^{14}	4818	2030	44578	12	445

Table 3.3 Results for the slotted ring example.

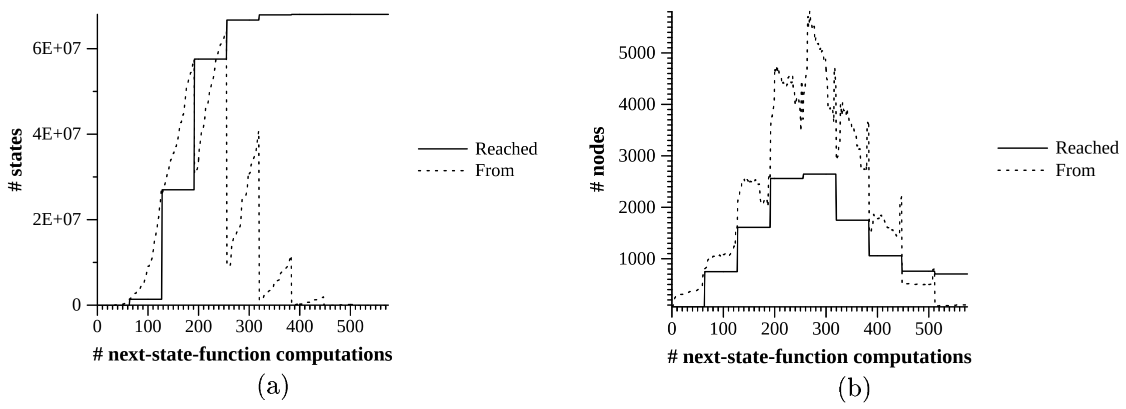


Figure 3.18 Symbolic traversal algorithm: (a) number of states calculated and (b) size of the variables after each computation of δ (for the 8-node slotted ring example).

3.9.2 Signal Transition Graphs

Several examples have been used to evaluate the efficiency of the proposed algorithms. Most STG examples are also scalable, thus the number of states can be exponentially increased by iteratively repeating a basic pattern.

Table 3.4 shows the obtained results. CPU time for each algorithm is presented. First, STG traversal and consistent state assignment are executed simultaneously (T+C). Next, non-input persistence (NI-p) and CSC are verified by using the set of reachable states. The last column

Example	n	# of places	# of signals	# of states	BDD size peak	BDD size final	CPU (seconds) T+C	NI-p	CSC	Total
master-read	-	36	13	8932	437	225	1	0	0	1
n-dining	10	90	30	6.0×10^7	2134	913	34	3	14	51
philosophers	20	180	60	3.7×10^{15}	8557	2019	765	142	18	927
	30	270	90	2.2×10^{23}	28002	3381	3296	551	45	3897
n-stage	30	120	30	6.0×10^7	7897	4784	132	0	38	170
Muller's	45	180	45	6.9×10^{11}	23590	10634	740	0	120	860
pipeline	60	240	60	8.4×10^{15}	53446	18788	3210	0	315	3525
n-user	20	81	40	2.2×10^7	1688	1688	9	2	2	13
DME	40	161	80	4.5×10^{13}	6568	6568	82	17	16	117
arbiter	60	241	120	7.0×10^{19}	14648	14648	286	56	56	403

Table 3.4 STG verification results.

shows the sum of all partial times. Since the *master-read* and *Muller's pipeline* examples are marked graphs, and hence they have no conflict places, the CPU time to check persistence and commutativity is negligible. The BDD sizes reported in Table 3.4 correspond to the size of the `Reached` set in the traversal algorithm. The number of variables of the BDD is the number of places plus the number of signals. The results show how STGs with a high degree of parallelism and a vast state space can be verified in moderate CPU times.

3.10 CONCLUSIONS

We have proposed a combination of boolean reasoning and BDD algorithms to manage the state explosion produced in Petri net analysis. It has been shown that BDDs can represent large sets of markings (10^{42} in the example) with a small number of nodes (2500). In addition, once the reachable markings have been generated, many properties can be verified in a straightforward manner.

The ordering of variables is a topic of major interest that must be studied in order to reduce even more the size of the BDDs, thus speeding up BDD operations.

4 ASYNCHRONOUS CIRCUIT VERIFICATION

Mentir bé és un art molt difícil, que poques persones arriben a practicar amb solvència i dignitat. Abunden els mentiders, però en general són mals mentiders: se'ls coneix que menteixen.

– Joan Fuster, "Diccionari per a ociosos".

About this chapter

This chapter presents how a circuit can be proved to be speed-independent under a certain environment. The circuit is composed with a Petri net describing the environment, and the set of reachable states of that closed system is computed. Both circuit and environment are modeled by means of boolean functions, that can be efficiently manipulated by using BDDs. The notion of failure state is given and it is shown that a circuit conforms to its specification, i.e. it is speed-independent with that particular environment, if none of the reachable states is a failure state. A sequence of events that produces a failure state, intended to help designers, is provided when the circuit is not found to be speed-independent. In addition to speed-independence verification, algorithms are provided to prove several properties.

Two approaches are described: a flat and a hierarchical one. The second method, more efficient, is proved to be exact. The demonstration of this theorem, in section 4.7.4 can be skipped in first reading. Finally, the features of the presented verification methods are illustrated through a benchmark suite, as well as, by the comparison to other authors.

4.1 INTRODUCTION

Traditionally, asynchronous circuits have been considered difficult to verify. When the circuit components are switching concurrently, the number of execution paths can be very large because of the variation of the component delays. Thus, a proper circuit behavior must be assured for all the possible execution paths. Since the number of feasible executions may be exponential in the number of components, it is desirable to automate the verification process, otherwise designers would probably be unable to face the problem.

4.1.1 Previous work

The verification of asynchronous circuits has been studied by several authors with different approaches. In this section, some of the most relevant efforts related with the verification of speed-independence and closest to the approach described in this chapter are presented.

When using *theorem proving* [Gor85], the asynchronous system and the specification are modeled in an appropriate logic and a proof is built as the circuit implies the specification. Although this is a flexible and powerful approach, the procedure is difficult to automate and might need to demonstrate a great number of theorems, which makes this methodology inefficient in practice, even for designers with good mathematical skills.

The following approaches are included in what has been called *model checking*, i.e. a description of the circuit (often complex) is checked to satisfy the specification of the circuit, usually expressed with some clear and concise formalism. Model checking was originally introduced in [CE81, QS81]. The original method consisted in building the state graph and verifying properties of the system by using *temporal logic*.

A common problem when using the state graph is the state explosion problem, i.e. the number of states grows exponentially with the size of the system. Burch et al. [CBM89, BCL$^+$94] proposed using Binary Decision Diagrams (BDDs) [Bry86] to represent the state graph, introducing *symbolic model checking*, much more efficient than the previous approach.

Other authors modeled circuit and specification as separated automata that interact with each other. In [Kur86], *language containment* techniques are proposed to verify that the language generated by the circuit is included in the language of the specification. Later, the same author proposed *homomorphic reductions* to simplify the problem [Kur87].

Trace theory [vdS83, Dil89] has been used to keep the history of the system. Then, properties of the system can be verified on the state graph by using temporal logic. In his thesis [Dil89], Dill already proposed hierarchical verification of speed-independence: if a component *conforms to* a trace structure, the behavior of that component can be safely substituted by the trace structure. However, this approach requires the designer to identify the basic components of the circuit and know their expected behavior in advance. This makes the approach impractical when a flat netlist of gates, with no explicit hierarchical organization, must be verified. Following Dill's approach, a subset of gates of the flat circuit, potentially substitutable by a complex gate, should be substituted by an equivalent trace structure. Not knowing how the environment of the complex gate will be inside the circuit, the trace structure should consider all possible input/output transitions and, therefore, include the state of all internal signals, which would preclude the subset of gates to be handled as a complex gate.

More recently, [McM92] has modeled both circuit and specification as Petri nets [Pet62], mitigating the state explosion problem by means of Petri net unfolding. Similarly, [HHY92] also represents circuit and specification as Petri nets, but the states are represented with BDDs and temporal logic is used to check some properties.

Beerel et al. [BBM94] propose a two-step approach. After verifying the circuit is *complex-gate equivalent* to its specification, hazard-freeness is verified by subsequently checking the monotonicity and acknowledgment of all signal transitions. A cube approximation that overestimates the set of states of the circuit is proposed to conservatively prove the absence of hazards. Although never found in the examples presented by the authors, *false negatives* are theoretically possible. Other limitations of this approach are that it is limited to externally-cut circuits (all memory elements must appear in the specification) and that the specification of the circuit is not allowed to express output choice (arbitration).

Kishinevsky et al. [KKTV94a, KKTV94b] presented a polynomial algorithm to verify a subclass of speed-independence, *distributivity*, from circuit behaviors described by *signal graphs*. The main limitation of their approach is that the signal graph must specify the transitions of all signals of the circuit and that neither choice nor nondeterminism are allowed in the signal graphs.

4.1.2 Overview

The goal of circuit verification is twofold:

1. on the one hand, the circuit has to be proved to conform to its specification, which means that no "bad" behavior is observed, and
2. on the other hand, the circuit must fulfill some properties, i.e. some "good" behavior is produced.

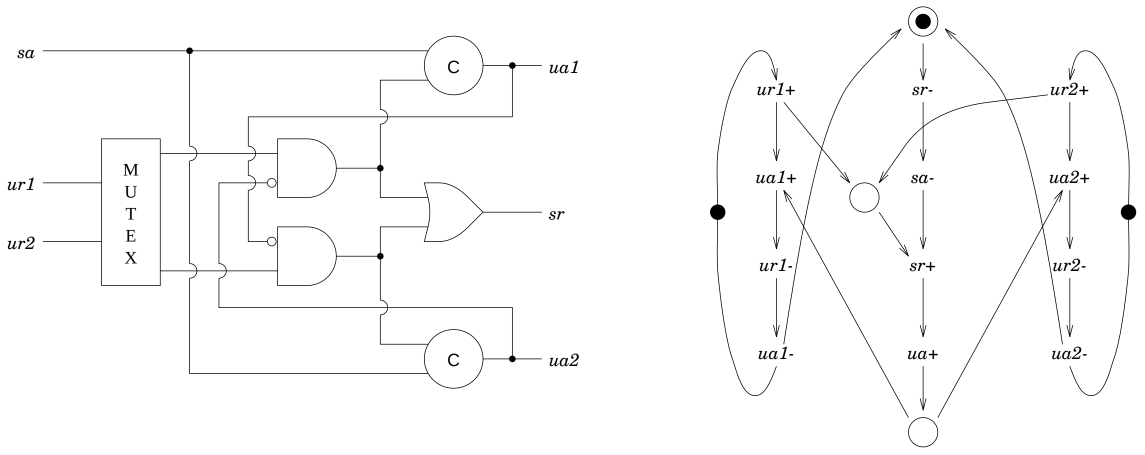


Figure 4.1 Verification scenario: a circuit composed with an environment.

The first part will construct the reachability set of a circuit when a certain environment, described as a Petri net, is assumed. Figure 4.1 depicts an example of such a verification scenario. Circuit correctness will be equivalent to say that none of the reachable circuit states is a failure state, as it will be described in section 4.5. A first approach is presented in section 4.6 and it is further refined in section 4.7.

The second part, verification of properties, is performed once the reachability set has been computed, although some properties can be checked as that set is being built. Algorithms for verifying several properties are given in section 4.8.

4.2 NOTATION AND TERMINOLOGY

A *circuit* is a triple $\langle \mathcal{V}, \mathcal{F}, s^0 \rangle$, where

$\mathcal{V} = \{v_1, \ldots, v_n\}$ is a set of signal variables (also referred simply as signals or variables),
$\mathcal{F}$ maps each signal $v_i \in \mathcal{V}$ to an n-variable boolean function, f_{v_i}, that represents the function computed by the gate that drives v_i, and
s^0 is the initial state of the circuit.

We will denote by $\mathcal{V}_I$ and $\mathcal{V}_G$ the set of signals corresponding respectively to input and gate signals. When necessary, we will differentiate the set of internal or hidden gate signals, $\mathcal{V}_H$, and the set of output gate signals, $\mathcal{V}_O$. Therefore, $\mathcal{V}_G = \mathcal{V}_H \cup \mathcal{V}_O$ and $\mathcal{V} = \mathcal{V}_I \cup \mathcal{V}_G$.

The *fanin* of signal $v_i \in \mathcal{V}$, denoted $\mathit{fanin}(v_i) \subseteq \mathcal{V}$, is the set of signals that f_{v_i} depends on. For sequential gates that, $v_i \in \mathit{fanin}(v_i)$. The *fanout* of signal $v_i \in \mathcal{V}$, $\mathit{fanout}(v_i) \subseteq \mathcal{V}$, is the set of signals that depend on v_i, i.e. $\mathit{fanout}(v_i) = \{v_k \in \mathcal{V} | v_i \in \mathit{fanin}(v_k)\}$.

For the sake of simplicity, in some sections we will assume the circuits to be *autonomous*. An autonomous circuit has no inputs, hence its evolution only depends on its own state. This fact simplifies notation, but the results can be extended to circuits with interface.

Given an autonomous circuit $C = \langle \mathcal{V}, \mathcal{F}, s^0 \rangle$, let $s[v\rangle s'$ denote the fact that state s' can be reached from s by switching $v \in \mathcal{V}$. The set of reachable states of C is denoted $[s^0\rangle$.

The state graph of an autonomous circuit $C = \langle \mathcal{V}, \mathcal{F}, s^0 \rangle$ is a state graph, $SG(C) = \langle S, E, V \rangle$, such that

$$\begin{array}{rcl} S & = & [s^0\rangle, \\ E & = & \{(s, s') \in S \times S \mid \exists v \in \mathcal{V} \text{ with } s[v\rangle s'\} \text{ and} \\ V & = & \mathcal{V}. \end{array}$$

Normally, single states will be denoted in lowercase letters, whereas capital letters will denote sets of states. Superindices may be used to distinguish different states, s, s', s^1, s^k, etc, or different sets of states, S, S', S^1, S^k, etc. Provided an encoding for states with n variables, s_i indicates the ith variable of that encoding, i.e. it is assumed that $s = (s_1, \ldots, s_n)$.

4.3 SPEED-INDEPENDENT CIRCUIT MODELING

In clocked digital systems, the state is determined by the value of the so called state variables. The order of the transitions along the combinational logic is not relevant, and the only restriction is that those transitions must occur within the clock period. In contrast, all the transitions in an asynchronous circuit have a meaning, and therefore, hazards, i.e. undesired or spurious signal transitions, must be avoided. Since all feasible execution paths have to be explored to detect possible hazards, the state of an asynchronous circuit will depend on all the signals.

We model a particular class of asynchronous circuits, *speed-independent* circuits, which correctly operate regardless of the delays of their components. Speed-independent circuits are modeled following the *unbounded gate delay* model [MB59]. Under such a model, delay elements are attached only to gate outputs and the delay magnitude is positive and finite, but unknown. The delay type we assume is *inertial* delay, i.e. pulses shorter than the delay magnitude are filtered out. Following this circuit model, the next state depends only on the present state, since once a gate is excited, that gate will eventually switch in the future.

Even this model assumes all forks to be isochronic, it is worth mentioning that non-isochronic forks can be also taken into account. One single variable is enough to represent the state of an isochronic fork, whereas non-isochronic forks need as many variables as the fanout of the gate driving the fork. Non-isochronic forks at primary inputs can be modeled by placing a buffer between the primary input and its fanout gates, as it is graphically depicted in figure 4.2(a). If the non-isochronic fork is driven by a gate v_i, then the fork can be modeled, as figure 4.2(b) illustrates, by replicating the gate v_i as many times as *fanout*(v_i). Therefore, modeling delay-insensitive and quasi-delay-insensitive circuits is also possible by just preprocessing the circuit netlist.

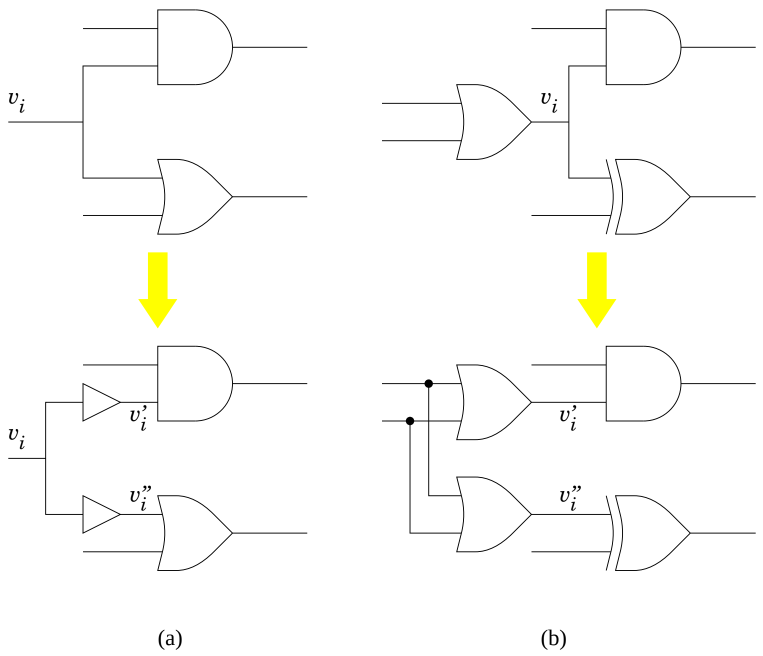


Figure 4.2 Modeling non-isochronic forks at (a) primary inputs and (b) gate outputs.

The states of a speed-independent circuit can be represented by boolean functions, with one boolean variable for each signal. We use v_i to indistinctly denote a circuit signal and the variable that represents that signal. The state of a circuit with n signals ($n = |\mathcal{V}|$) is determined by the value of its signals and that state can be represented by a minterm of an n-variable logic function. That minterm is the characteristic function of a state of the circuit. Sets of states can be represented as the disjunction of the minterms representing those states. A circuit with n binary signals can have as many as 2^n states, and we will refer to $\mathcal{S}_\mathcal{V}$ as the set of all possible states of a circuit with the set of signals $\mathcal{V}$. The set of all possible subsets of $\mathcal{S}_\mathcal{V}$ is denoted $2^{\mathcal{S}_\mathcal{V}}$.

4.3.1 Gate modeling

Let us recall the gate model for speed-independent circuits (section 1.3, Muller model), in which each gate is modeled as an instantaneous logic function followed by an unbounded inertial delay. Henceforth a "gate" will denote both the function and its associated delay. Two different types of gates must be considered: *combinational* and *sequential* gates, also called *memory elements*. An example of combinational gate can be found in figure 4.3(a). Under the unbounded gate delay model, sequential gates are such that the delay output is connected to one of the gate inputs, as depicted in figure 4.3(c). The sequential gate in figure 4.3(c) is equivalent to the symbol of the C element in figure figure 4.3(b). Typically, when a sequential gate has an associated symbol, the implicit feedback loop does not appear for simplicity. Without loss of generality, only one-output gates will be considered, since n-output gates can be represented as n one-output gates.

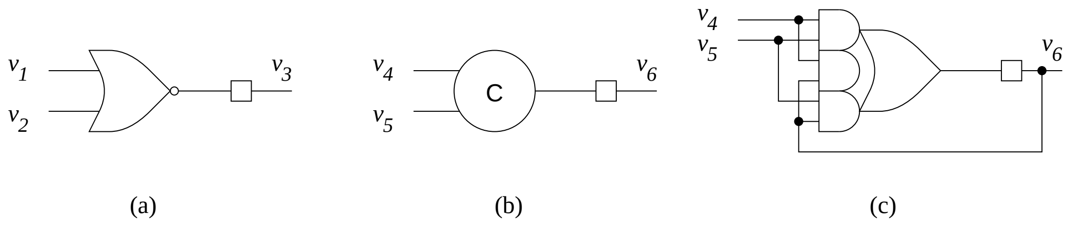


Figure 4.3 (a) Combinational NOR gate. (b) C memory element and (c) its internal representation.

In our framework, gate switching is also simulated with boolean functions. We will indistinctly use v_k to denote a circuit signal and the gate driving that signal. Let us assume that gate v_k implements the function $f_{v_k}(v_1, \ldots, v_j)$. Gate v_k is said to be sequential only if $v_k \in \mathit{fanin}(v_k)$, i.e. $v_k \in \{v_1, \ldots, v_j\}$, and combinational otherwise. A gate v_k is said to be *excited* when $v_k \neq f_{v_k}(v_1, \ldots, v_j)$, and *stable* otherwise. We represent the set of states in which a gate is excited and the output will eventually become 1 by the *positive excitation function*, f^+:

$$f^+_{v_k}(v_1, \ldots, v_j) = \bar{v}_k \cdot f_{v_k}(v_1, \ldots, v_j) \ ,$$

Similarly, we can define the *negative excitation function*, f^-, as follows:

$$f^-_{v_k}(v_1, \ldots, v_j) = v_k \cdot \overline{f_{v_k}(v_1, \ldots, v_j)} \ .$$

These definitions are analogous to the flow tables presented in [DC86]. Other authors have proposed to model gates with Petri nets [McM92, HHY92]. However, each gate may result in a net with several places and transitions that would cause a more complex model for verification. The model proposed in this chapter, two excitation functions per gate, is more efficient. Next we show, as examples, those characteristic functions for the NOR gate and the C element in figure 4.3:

$$v_3 = f_{v_3}(v_1, v_2) \quad = \overline{v_1 + v_2} \qquad \begin{cases} f^+_{v_3}(v_1, v_2, v_3) = \bar{v}_3 \cdot \bar{v}_1 \cdot \bar{v}_2 \\ f^-_{v_3}(v_1, v_2, v_3) = v_3 \cdot (v_1 + v_2) \ , \end{cases}$$

$$v_6 = f_{v_3}(v_4, v_5, v_6) \quad = v_4 \cdot v_5 + v_6 \cdot (v_4 + v_5) \qquad \begin{cases} f^+_{v_6}(v_4, v_5, v_6) = \bar{v}_6 \cdot v_4 \cdot v_5 \\ f^-_{v_6}(v_4, v_5, v_6) = v_6 \cdot \bar{v}_4 \cdot \bar{v}_5 \ . \end{cases}$$

The *image computation function*, $\delta_G : 2^{\mathcal{S}_\mathcal{V}} \times \mathcal{V}_G \longrightarrow 2^{\mathcal{S}_\mathcal{V}}$, is a function that given a set of states S and a non-input signal v returns those states that can be reached by switching v. Formally:

$$\delta_G(S, v) = \{s' \mid \exists s \in S \text{ with } s[v\rangle s'\} \ .$$

The function δ_G can be computed by using excitation functions as follows:

$$\delta_G(S, v) = (S \cdot f^+_v)\downarrow_{\bar{v}} \cdot v + (S \cdot f^-_v)\downarrow_v \cdot \bar{v} \ .$$

To illustrate this, we will calculate the new set of states S' reached by switching the NOR gate in figure 4.3(a) from the set of states S. Let us assume that

$$S = v_1\bar{v}_2v_3\bar{v}_4v_5 + \bar{v}_1v_2\bar{v}_3v_4\bar{v}_5 + \bar{v}_1\bar{v}_2\bar{v}_3\bar{v}_4v_5 \ .$$

The product of S by the excitation functions of the gate $(f^+_{v_3}, f^-_{v_3})$:

$$\begin{aligned} S \cdot f^+_{v_3} &= \ S \cdot \bar{v}_3 \cdot \bar{v}_1 \cdot \bar{v}_2 &&= \bar{v}_1\bar{v}_2\bar{v}_3\bar{v}_4v_5 \ , \\ S \cdot f^-_{v_3} &= \ S \cdot v_3 \cdot (v_1 + v_2) &&= v_1\bar{v}_2v_3\bar{v}_4v_5 \ , \end{aligned}$$

gives the states in which the gate is excited. The following operations simulate, respectively, the rising and falling of signal v_3:

$$\begin{aligned}(S \cdot f^{+}_{v_3})\downarrow_{\bar{v}_3} \cdot v_3 &= \bar{v}_1\bar{v}_2\bar{v}_3\bar{v}_4 v_5 \downarrow_{\bar{v}_3} \cdot v_3 = \bar{v}_1\bar{v}_2 v_3\bar{v}_4 v_5 \ , \\ (S \cdot f^{-}_{v_3})\downarrow_{v_3} \cdot \bar{v}_3 &= v_1\bar{v}_2 v_3\bar{v}_4 v_5 \downarrow_{v_3} \cdot \bar{v}_3 = v_1\bar{v}_2\bar{v}_3\bar{v}_4 v_5 \ .\end{aligned}$$

Finally, the set of states S' is computed as the union of the states where signal v_3 has already risen or fallen:

$$S' = (S \cdot f^{+}_{v_3})\downarrow_{\bar{v}_3} \cdot v_3 + (S \cdot f^{-}_{v_3})\downarrow_{v_3} \cdot \bar{v}_3 = \bar{v}_1\bar{v}_2 v_3\bar{v}_4 v_5 + v_1\bar{v}_2\bar{v}_3\bar{v}_4 v_5 \ .$$

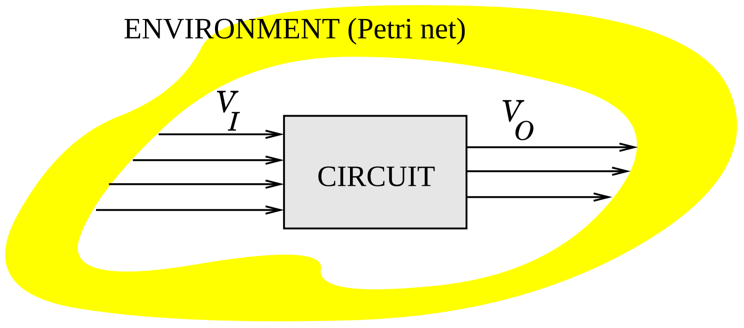


Figure 4.4 Environment-circuit system.

4.4 ENVIRONMENT AND CIRCUIT COMPOSITION

As shown in figure 4.4, the verification scenario consists of a closed system in which a circuit is composed with its specification or environment. This environment could be described in many different ways, e.g. process algebras [Ebe89], CSP [Hoa78] or Petri nets [Pet62]. Petri nets are a powerful formalism for specifying asynchronous circuits, very popular in the asynchronous circuit design community. In addition, there are several methodologies that use Petri nets as circuit specification [RY85, Chu87, Van90, Lav92, Pas96]. Thus, it is very attractive to use the same formalism for describing a circuit to be synthesized and afterwards for verifying that circuit against its specification. Examples of a circuit and its Petri net specification can be found in figures 4.1 and 4.6.

Given a Petri net $\langle \mathcal{P}, \mathcal{T}, \mathcal{F}, m_0 \rangle$ specifying the environment that interacts with a circuit, there is a relationship between the circuit interface signals and some Petri net transitions. We denote by T_{v+} (T_{v-}) the set of transitions in the Petri net that specify a rising (falling) transition of signal v. We use T_{v*} to denote either T_{v+} or T_{v-}. We will consider that $\mathcal{T} = \mathcal{T}_I \cup \mathcal{T}_O \cup \mathcal{T}_H$, where $\mathcal{T}_I$ and $\mathcal{T}_O$ are respectively the sets of transitions associated to input and output signals, while $\mathcal{T}_H$ is the set of internal Petri net transitions, i.e. transitions that are not associated to any circuit signal.

The set of all possible states of an environment-circuit system is a subset of the Cartesian product of the possible sets of states of each subsystem, $\mathcal{S} = \mathcal{M}_{\mathcal{P}} \times \mathcal{S}_{\mathcal{V}}$. Therefore, if $m = |\mathcal{P}|$ and $n = |\mathcal{V}|$, the state of such a system is defined by the ordered pair

$$((p_1, \ldots, p_m), (v_1, \ldots, v_n)) \ ,$$

where $(p_1, \ldots, p_m)$ is a marking of the Petri and $(v_1, \ldots, v_n)$ represents a state of the circuit. Thus, states of an environment-circuit system can be represented as minterms of $n + m$ variables, and sets of states as the sum of the minterms of the states in the set.

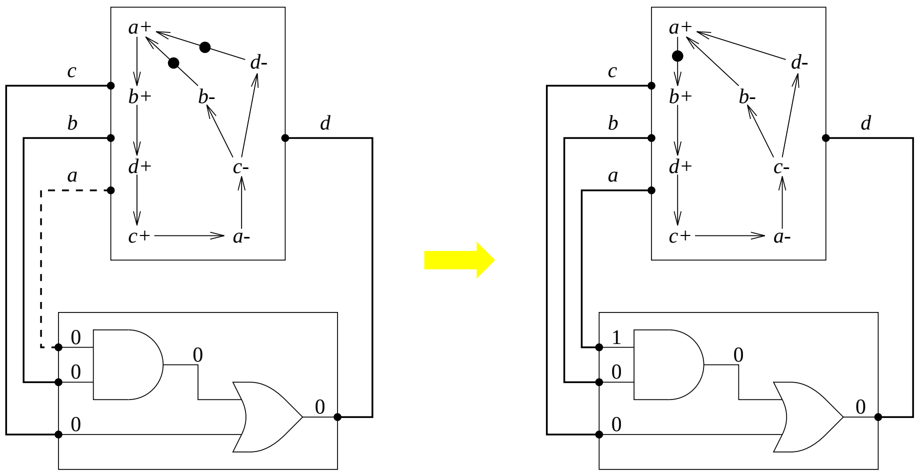


Figure 4.5 Transition $a+$ is synchronized with switching a from 0 to 1.

The previously defined image computation formulae, δ_N (see section 3.2.2) and δ_G, can be extended for the environment-circuit system as:

$$\begin{aligned} \delta_N &: 2^{\mathcal{S}} \times \mathcal{T} \longrightarrow 2^{\mathcal{S}} \quad \text{and} \\ \delta_G &: 2^{\mathcal{S}} \times \mathcal{V}_G \longrightarrow 2^{\mathcal{S}} \ . \end{aligned}$$

These functions can be used for image computation of internal environment transitions and internal gate switching. However, new functions have to be defined for interface signals, in order to simulate the synchronization between Petri net and circuit. Figure 4.5 depicts a synchronized change in both subsystems. For transitions associated to input and output circuit signals the image computation functions are, respectively:

$$\begin{aligned} \delta_I &: 2^{\mathcal{S}} \times \mathcal{T}_I \longrightarrow 2^{\mathcal{S}} \quad \text{and} \\ \delta_O &: 2^{\mathcal{S}} \times \mathcal{T}_O \longrightarrow 2^{\mathcal{S}} \ . \end{aligned}$$

The environment "decides" when an input signal of the circuit has to switch. Thus, when a transition in T_{v*} is fired, signal v must switch accordingly. The image computation function for STG transitions (see section 3.3.2) can be used to fire transitions associated to input signals:

$$\delta_I(S,t) = \delta_S(S,t) = \begin{cases} \delta_N(S,t)\downarrow_{\bar{v}} \cdot v \ , & \text{if } t \in T_{v+} \\ \delta_N(S,t)\downarrow_{v} \cdot \bar{v} \ , & \text{if } t \in T_{v-} \end{cases} \ .$$

In the case of output signals, the circuit takes the initiative of the change. Given a transition associated to an output signal, i.e. $t \in T_{v*}$, with $v \in \mathcal{V}_O$, the function δ_O performs image computation as follows:

$$\delta_O(S,t) = \delta_N(\delta_G(S,v),t) \ .$$

Note that if more than one transition $t \in T_{v*}$ is enabled in a given state, it may indicate nondeterminism or a bad environment specification. This can be reported as a warning.

4.5 VERIFICATION OF SPEED-INDEPENDENCE

Speed-independence is not a property of a circuit by itself but of the behavior of a circuit under a certain environment. In our framework, the behavior of the environment will be represented by a Signal Transition Graph (STG) [RY85, Chu87] in which the outputs of the STG will be

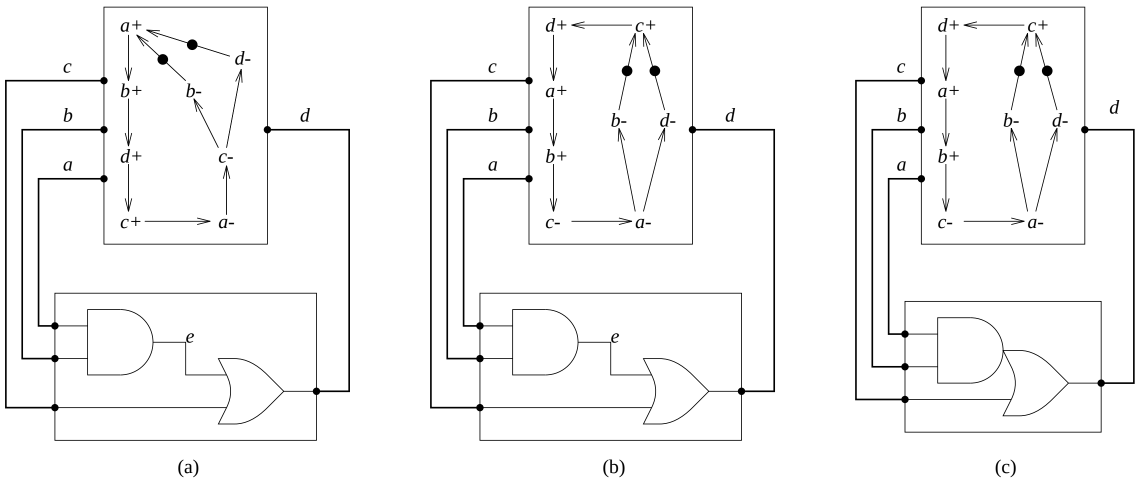


Figure 4.6 (a) Circuit with a speed-independent behavior, (b) The same circuit with a hazardous behavior, (c) Equivalent speed-independent complex-gate circuit.

inputs of the circuit and vice versa. Figures 4.6(a) and 4.6(b) depict a circuit excited by two different environments. The circuit is speed-independent with environment (a), but hazardous with environment (b). In the latter case, a static hazard can be produced on signal d when, in the state $abcde = 11110$, the event $c-$ arrives before e has switched to 1. However, note that an equivalent complex-gate implementation of the same circuit (figure 4.6(c)) can be hazard-free.

Figure 4.7 shows the state graph obtained by a reachability analysis of the system in figure 4.6(a), in which place variables do not appear for simplicity. Verification through reachability analysis [BCL$^+$94] will check that each transition produced at the outputs of the circuit can be accepted by the environment, i.e. a transition with the same label is enabled in the STG.

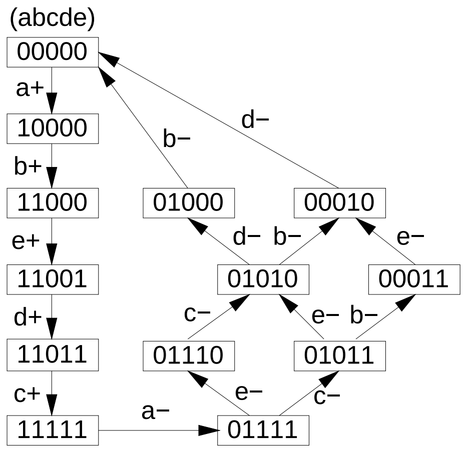


Figure 4.7 State Graph of the circuit in figure 4.6(a) (without place variables, for clarity).

4.5.1 Failure states

The verification procedure checks that the events generated by the circuit, in response to the stimuli dictated at its inputs, can be accepted by the environment. Those states in which there is a signal $v \in \mathcal{V}_O$ positively (negatively) excited while no transition $t \in T_{v+}$ (T_{v-}) is enabled are called *failure states*. These states model situations in which the circuit generates a signal transition not expected by the environment (see example in figure 4.8).

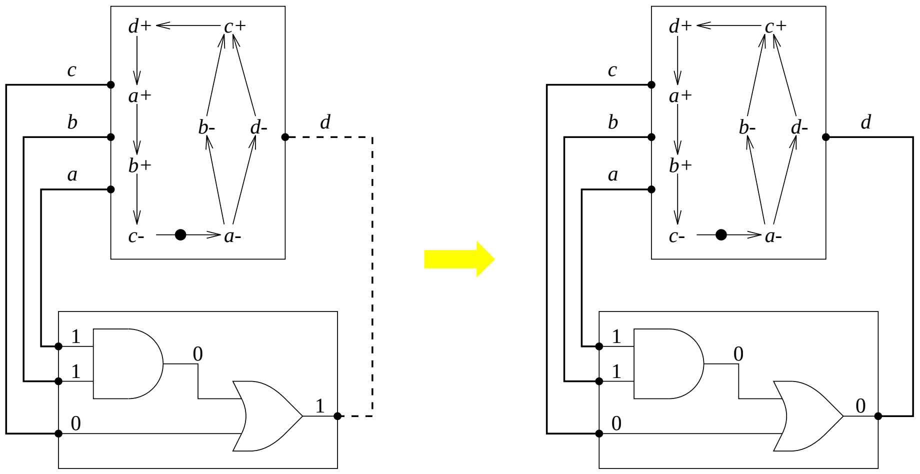


Figure 4.8 Switching signal d from 1 to 0 is not accepted by the environment.

Failure states are produced when the circuit does not conform to its specification. This may be due to a bad synthesis procedure that causes the circuit to have a hazardous or non-semi-modular behavior. Also, if a circuit is verified (by mistake) against the specification of another circuit, e.g. a nuclear reactor controller is verified against the specification of a kettle controller, failure states will surely appear. In any case, an error in a set of states S is detected when any of the following equations is satisfied:

$$\bigcup_{t \in T_{v+}} S \cdot f_v^+ \cdot \overline{\mathrm{E}_t} \neq \emptyset \ , \qquad \bigcup_{t \in T_{v-}} S \cdot f_v^- \cdot \overline{\mathrm{E}_t} \neq \emptyset \ .$$

Note that function δ_O is correctly defined only if no failure states are contained in S, therefore malfunction detection must be done before $\delta_O(S,t)$ is computed.

A circuit with a given environment will be speed-independent if no failure state is present in the set of reachable states of the environment-circuit system. This is equivalent to say that the circuit conforms to its specification.

4.5.2 Semi-modularity

Semi-modularity is a more robust concept than speed-independence, but both concepts are tightly related for most practical cases, as subsequently explained (see [YLSV92] for further details). In addition to conformance to the environment, semi-modularity can be checked at each image computation step, and violations of this property can also be reported as errors.

Definition 4.1 (Semi-modularity) *A signal v_i is* semi-modular *with respect to signal $v_k \in$ fanin(v_i) $(v_i \neq v_k)$ if the gate that drives v_i, having been excited, cannot become stable by changing the value of v_k. Formally,*

$$s[v_k\rangle s' \Longrightarrow [s_i \neq f_{v_i}(s) \Longrightarrow s'_i \neq f_{v_i}(s')] \ .$$

A signal v_i is semi-modular if it is semi-modular with respect to all its fanin signals. A circuit is semi-modular if all its signals are semi-modular.

Definition 4.2 (Strongly-live circuit) *A circuit is* strongly live [YLSV92] *iff its state graph is strongly connected and for each signal v_i there exists a state $s \in S$ in which v_i is excited.*

Theorem 4.1 ([YLSV92]) *If a circuit is* strongly live*, then the circuit is speed-independent iff it is semi-modular.*

4.6 FLAT VERIFICATION OF SPEED-INDEPENDENT CIRCUITS

Formally verifying that a circuit is speed-independent under a certain environment can be done by building a closed environment-circuit system, computing all the reachable states of the system and proving that none of them is a failure state. Then we can say that the circuit *conforms to the environment* or that the circuit is a speed-independent implementation of its specification.

Since the number of states can be exponential in the number of variables, explicit enumeration techniques will result unfeasible even for circuits with just a few gates. On the contrary, sets of states can be represented by their characteristic functions and, hence, BDDs can be used to efficiently represent and manipulate such functions.

As a first approach, circuit and environment will be represented by using all their variables: places and signals. Although this *flat verification* is an efficient approach itself, in further sections it will be shown how it can be done even better.

The set of reachable states can be calculated by using a symbolic algorithm, similar to the one described in section 3.5 for traversing Petri nets. The basic algorithm, depicted in figure 4.9, works as follows. Given the initial state (or states) S^0, at iteration i, the set of states that can be reached after i transitions is computed. At the ith iteration three different set variables can be distinguished:

- `Reached` contains the states visited until iteration $i-1$,
- `From` keeps the last set of states reached after exactly $i-1$ iterations and
- finally, `New` accumulates the states reachable from the states in `From` by firing one transition or by switching one signal.

The `New` set is calculated by successively applying the image computation functions described in previous sections. Prior to image computation, the existence or absence of failure states must be checked. Note that `New` may contain states not visited so far, as well as already visited states. The really new states, `New` $-$ `Reached`, constitute the `From` set for the next iteration, and the newly visited states are also added to `Reached`. Since the number of reachable states is finite, after a number of iterations a *fixed point*, is reached and the BFS algorithm finishes.

The basic BFS algorithm in figure 4.9 can be significantly improved by using similar techniques to those proposed in section 3.7.

4.6.1 Error diagnosis

Telling whether or not a circuit conforms to its specification is a useful information, but such a simple statement may not help designers to find out why their design is wrong. On the other hand,

```
basic_BFS_traversal {
   Reached := From := $S^0$;
   do {
      test_for_failure_states(From);
      New := $\emptyset$;
/* Let $\delta$ be either $\delta_G$, $\delta_I$, $\delta_O$ or $\delta_N$ */
/* depending on whether $x$ in $\mathcal{V}_H$, $\mathcal{T}_I$, $\mathcal{T}_O$ or $\mathcal{T}_H$ */
      for each signal/transition $x \in \mathcal{V}_H \cup \mathcal{T}$ {
         New := New $\cup$ $\delta$(From,$x$);
      }
      From := New - Reached;
      Reached := New $\cup$ Reached;
   } while (From $\neq \emptyset$);
}
```

Figure 4.9 Basic BFS algorithm.

an example of a situation producing an unexpected behavior can be of great help. The algorithm in figure 4.11 gives a sequence of events that produces a failure state, i.e. a *counterexample* showing when the circuit behavior is erroneous.

From a failure state, it is performed a backward traversal, restricted to the states that had been visited during the forward traversal, and a trace from the initial state until the failure state is given. To perform this backward traversal, we need to define the *preimage computation functions*. The preimage computation function for transitions is computed as follows:

$$\delta_N^{-1}(S,t) = (S\downarrow_{\mathrm{ASM}_t} \cdot \mathrm{NSM}_t)\downarrow_{\mathrm{NPM}_t} \cdot \mathrm{E}_t \ ,$$

that intuitively is equivalent to changing the direction of the arcs of the Petri net. A gate will switch backwards by changing the output value when it is stable and, therefore, becoming excited. Figure 4.10 illustrates how a stable gate switches backwards to an excited state. The preimage computation function of a signal is computed as:

$$\delta_G^{-1}(S,v) = \left(S \cdot f_v^1\right)\downarrow_v \cdot \bar{v} \ + \ \left(S \cdot f_v^0\right)\downarrow_{\bar{v}} \cdot v \ ,$$

where f_v^0 and f_v^1 respectively represent the states in which v is stable at 0 or 1, i.e. $v = f_v(v_1, \ldots, v_i)$. Function δ_G^{-1} changes v into $\bar{v}$ in those states in which the gate driving v is stable at 1, an vice versa for the states with v stable at 0. Given δ_N^{-1} and δ_G^{-1}, the definition of preimage computation functions for input and output signal transitions, δ_I^{-1} and δ_O^{-1}, is as follows:

$$\begin{aligned} \delta_I^{-1}(S,t) &= \begin{cases} \delta_N^{-1}(S,t)\downarrow_v \cdot \bar{v} \ , & \text{if } t \in T_{v+} \\ \delta_N^{-1}(S,t)\downarrow_{\bar{v}} \cdot v \ , & \text{if } t \in T_{v-} \end{cases} \quad \text{and} \\ \delta_O^{-1}(S,t) &= \delta_N^{-1}(\delta_G^{-1}(S,v),t) \ , \end{aligned}$$

provided that $t \in T_{v*}$. Note that some of the states returned by the different preimage computation functions may have not been visited, therefore it may be necessary to restrict those states to a set of reachable states.

The diagnosis algorithm on figure 4.11 prints a sequence of transitions from the initial state, s^0, until a failure state when `obtain_trace(Failure,Reached)` is called. We assure that the given trace will not be an impossible trace by restricting δ^{-1} to the reachable set of states, and by eliminating the visited states we ensure the algorithm to converge.

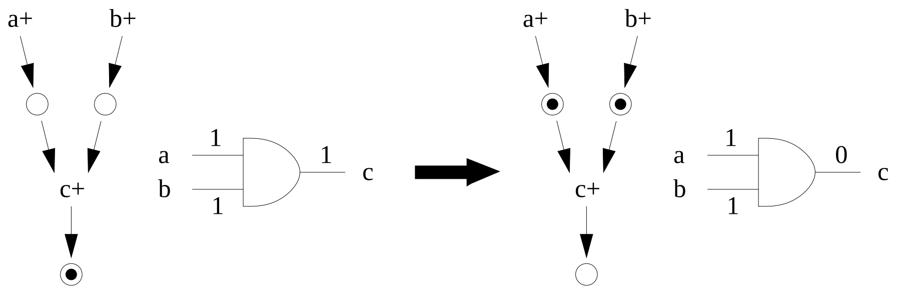


Figure 4.10 Stable AND gate becoming excited by backward switching.

```
obtain_trace (From,Reached) {
/* Let $\delta^{-1}$ be either $\delta_G^{-1}$, $\delta_I^{-1}$, $\delta_O^{-1}$ or $\delta_N^{-1}$ */
/* depending on whether $x$ in $\mathcal{V}_H$, $\mathcal{T}_I$, $\mathcal{T}_O$ or $\mathcal{T}_H$ */
   for each signal/transition $x \in \mathcal{V}_H \cup \mathcal{T}$ {
      Pre := $\delta^{-1}$(From,$x$) $\cap$ Reached;
      if (Pre $\neq \emptyset$) {
         if ( (Pre $\cap$ $s^0$) $\neq \emptyset$) then
            r := TRUE; /* trace found */
         else
            r := obtain_trace(Pre,Reached-Pre);
         if (r) then {
            print_transition(From,$x$);
            return r;
         }
      }
   }
   return FALSE;
}
```

Figure 4.11 Diagnosis algorithm.

4.7 HIERARCHICAL VERIFICATION OF SPEED-INDEPENDENT CIRCUITS

The complexity of verification increases with the number of circuit signals, and in speed-independent circuits all signals are relevant in order to keep the state of the circuit. An approach that considerably reduces the required number of variables is to be presented in this section. The technique consists of two steps such that:

1. At first, given a circuit C, most of its internal combinational signals are abstracted, thus obtaining a reduced circuit, $\widehat{C}$, formed by several complex gates. Such a reduction can be observed, for example, between figures 4.12(a) and 4.12(b). The reduced circuit $\widehat{C}$ is verified to satisfy the specification of the original circuit C.

2. Having proved that circuit $\widehat{C}$ is correct, the second step consists in verifying that C is also correct. As it will be demonstrated in section 4.7.4, the original circuit C is correct if and only if each complex gate in $\widehat{C}$ is semi-modular.

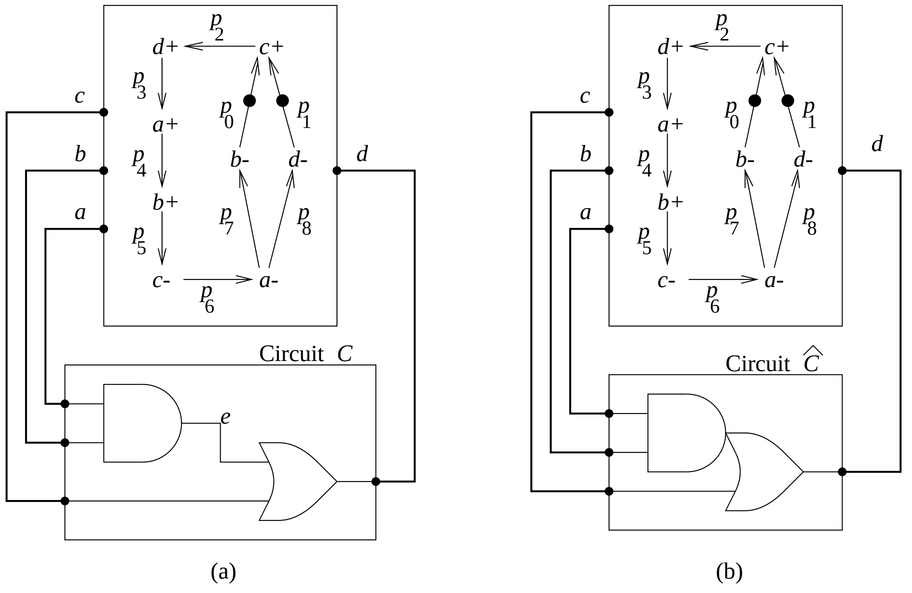


Figure 4.12 (a) Environment-circuit system and (b) its complex-gate equivalent system.

With hierarchical verification the number of variables relevant at each step of verification is drastically reduced: during verification of the complex gate equivalent circuit, only the input and output signals of complex gates are required; during semi-modularity checking of a complex gate only the interface and internal signals of the gate are needed. This last statement is true for most practical circuits, although, in some contrived circuits, this number may be slightly increased, as it will be explained in section 4.7.3. Since a critical factor that determines the complexity of verification is the number circuit signals, the reduction of the variables needed along the various algorithms results in a much more efficient verification method.

4.7.1 Functional and behavioral correctness

The two steps of hierarchical verification are tightly related with the concepts of *functional* and *behavioral correctness*.

Functional correctness: A circuit is said to be *functionally correct* if an appropriate combination of the delays of its components can produce the behavior expected by the environment.

Behavioral correctness: A functionally correct circuit is said to be *behaviorally correct* if it produces the behavior expected by the environment regardless the delay assigned to each component, provided that the delays are within the margins assumed for the delay model and the technology. For speed-independent circuits, delays are in the range $(0, \infty)$.

We can say now that the circuit in figure 4.12(a) is functionally correct, since by assigning the AND gate a delay shorter than the delay between the transitions $b+ \rightarrow c-$, the generated behavior is the one expected by the environment. However, this circuit is behaviorally incorrect, because long delays on the AND gate may produce a static hazard on d. The circuit in figure 4.12(b) is both functionally and behaviorally correct, as the complex-gate architecture assumes zero delay for the AND gate.

Verification of functional correctness

A speed-independent circuit must behave correctly for any finite delay of its components. A particular case consists in "moving" the delay of a gate to its fanout gates. Let us take as example the circuit in figure 4.12(a). We can move the delay of gate e to its fanout gate d, and we obtain the complex gate in figure 4.12(b). We refer to this kind of gate clustering as *collapsing*. Since all the delays are in the range $(0, \infty)$, the sum of the delays of d and e is still in the same range. The behavior of the output of a complex gate is included in the behavior of the original circuit (prior to collapsing).

In our framework, functional correctness is verified by collapsing some of the gates of the circuit. In this way, multiple gates can be collapsed into one complex gate and, thus, internal signals eliminated for the verification. Only for memory elements or outputs of complex gates, signals cannot be eliminated.

In the example of figure 4.12(a), functional correctness is verified by first collapsing the AND and OR gates into one complex AND-OR gate and eliminating signal e. The state graph of the system formed by circuit $\widehat{C}$ and its environment is depicted in figure 4.13(a). The state of such system is determined by the signals not abstracted plus the variables of the environment.

Functional correctness can be verified by an algorithm like the one used for flat verification (see figure 4.9). If the collapsed circuit $\widehat{C}$ reaches some failure state, the original circuit C will also enter some failure state, since the behavior of $\widehat{C}$ is included in that of C. On the contrary, if no failure states appear, it can be said that circuit $\widehat{C}$ satisfies the specification of the original circuit C. Verification of functional correctness becomes simpler and faster because of the elimination of internal signals. Moreover, design errors that do not depend on the delays of the gates can be detected soon, without requiring an exhaustive verification of the temporal relations among all signals.

Verification of behavioral correctness

The second step of verification is devoted to detect hazards inside the complex gates. Intuitively, this is performed as follows. Once verified that the collapsed circuit is speed-independent, the state graph obtained in functional correctness verification is projected onto the interface signals of each complex gate and taken as the environment of the complex gate. By traversing the gate with this environment, the state graph of the complex gate is built and semi-modularity verified. If all complex gates are semi-modular, the circuit is behaviorally correct, otherwise the circuit is functionally correct but behaviorally incorrect.

The state graph in figure 4.13(b) has been obtained by projecting the state graph in figure 4.13(a) onto the interface signals of the complex AND-OR gate, $\{a, b, c, d\}$. While the state graph of the complex gate (figure 4.13(c)) is being built, the non semi-modular state 11010 appears, since firing transition $a-$ makes the AND gate to become stable before its output e raises. If a violation of semi-modularity is detected, it is unnecessary to finish forming the rest of the graph and verification concludes that the circuit is behaviorally incorrect.

In general, large circuits will be collapsed into several complex gates. As illustrated in figure 4.14, this can be done hierarchically according to efficiency criteria for verification.

Isomorphic groups of gates can be mapped onto the same complex gate. In the example in figure 4.14 there is a pattern repeated twice: an OR gate which inputs are an AND gate and a primary input. We will show in section 4.7.5 that we can project the environment of several

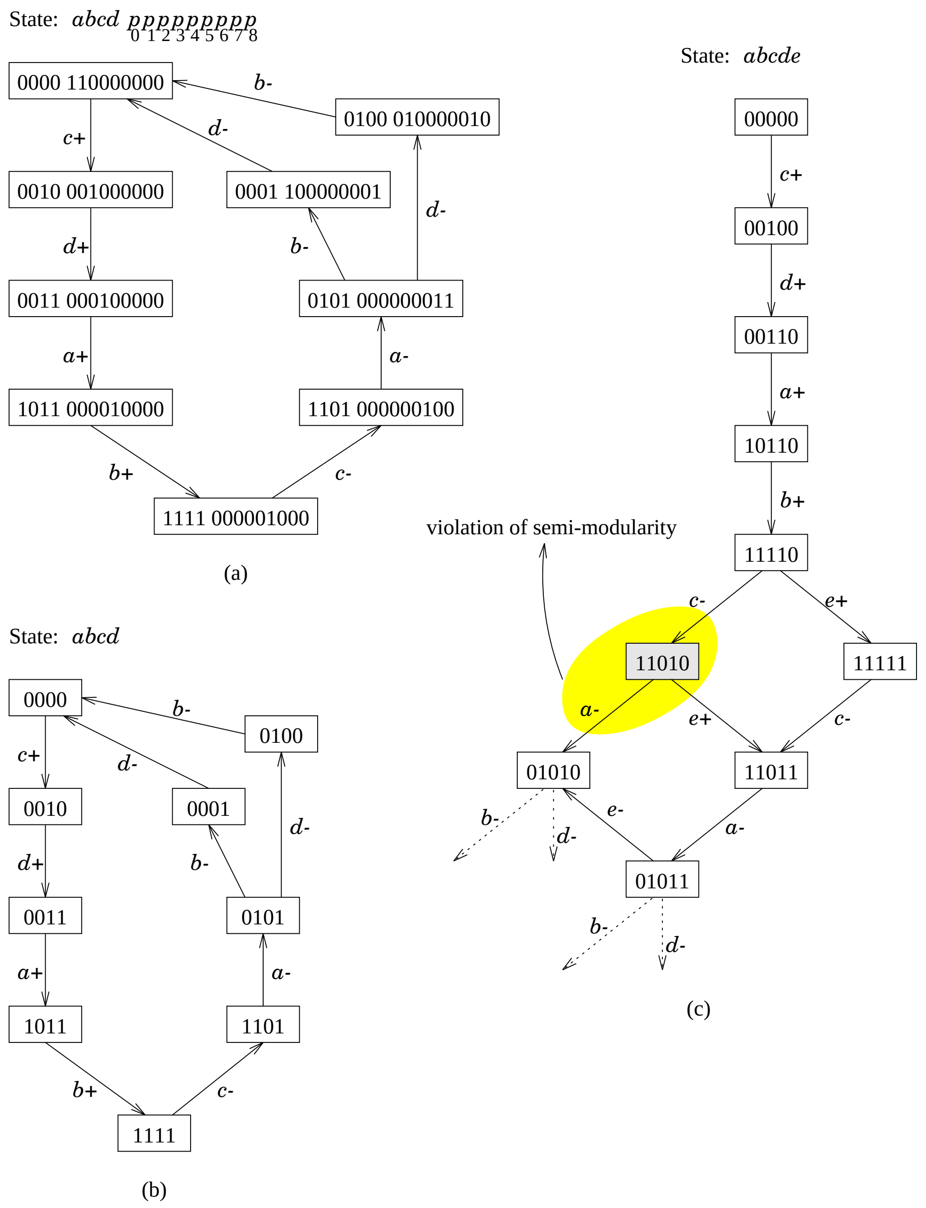


Figure 4.13 (a) State graph after functional correctness verification. (b) Projection onto the interface signals of the complex gate. (c) Part of the state graph after behavioral correctness verification.

complex gates onto one single state graph and verify their hazard-freeness at a time. This hierarchy allows us to verify isomorphic subcircuits together.

In case the complex gate were verified to be hazard-free, its corresponding state graph would be projected onto the input/output signals of its components and the same operation would be performed at the next level of the hierarchy.

In order to ease the understanding of the presented approach, a key issue has been intentionally kept hidden: projecting a state graph onto a subset of signals may map semantically different states onto a same state and, consequently, the structure of the graph may drastically change. In most practical cases, even with this apparent loss of information, behavioral correctness can

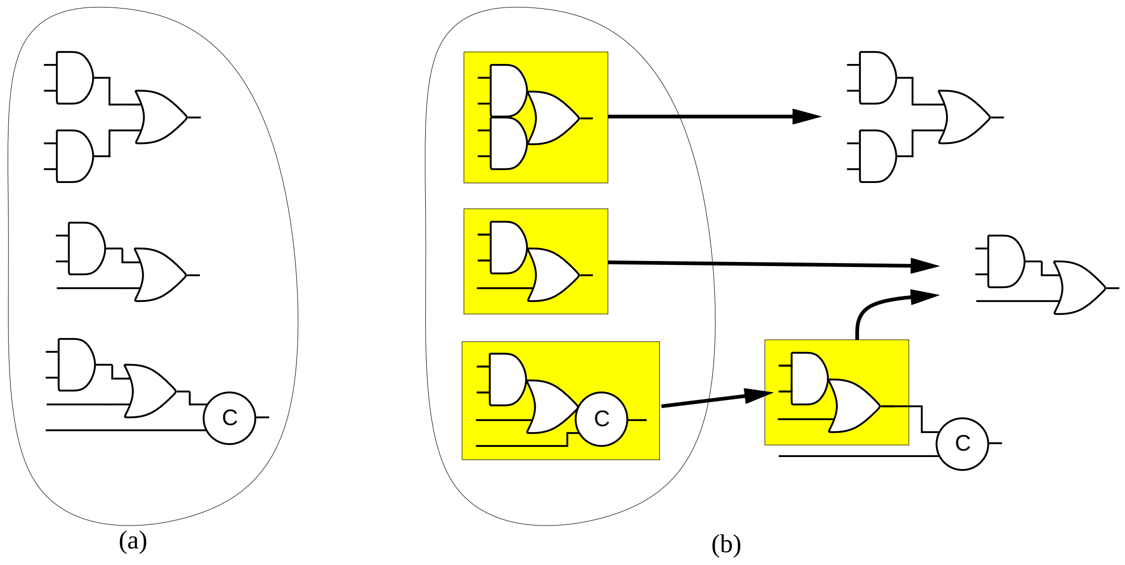


Figure 4.14 (a) Flat circuit (8 gates). (b) Hierarchical complex-gate organization (3 complex gates).

be *exactly* verified. How our method is still exact rather than conservative will be discussed in section 4.7.3.

4.7.2 Reduction to complex gates

This section provides the means that enable to eliminate some signals of a circuit to simplify its verification. We propose to collapse several gates into one complex gate with the same functional behavior and eliminate the internal signals.

Let us assume we have a circuit $C = \langle \mathcal{V}, \mathcal{F}, s^0 \rangle$ with signal v_n being driven by a combinational gate, i.e. $v_n \notin \mathit{fanin}(v_n)$. Given a state s, for any function f_{v_i} that does not depend on v_n it holds that

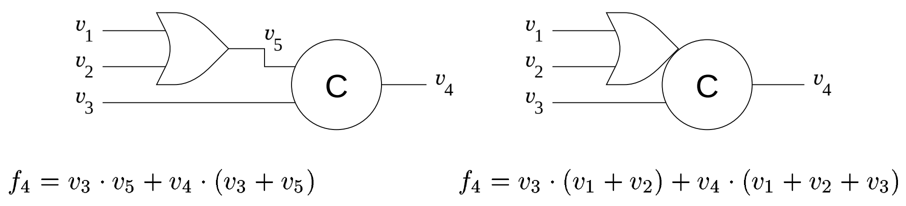
$$f_{v_i}(s) = f_{v_i}(s_1, \ldots, s_{n-1}, 0) = f_{v_i}(s_1, \ldots, s_{n-1}, 1) \ .$$

Let us build a new circuit $\widehat{C} = \langle X, \widehat{F}, \widehat{s}^0 \rangle$, with $X = \mathcal{V} - \{v_n\}$. Assuming $\widehat{s} = \mathit{proj}_X(s)$ and $1 \leq i < n$, the boolean expression of each gate in $\widehat{C}$ is defined as follows:

$$\widehat{f}_{v_i}(\widehat{s}) = \begin{cases} f_{v_i}(s_1, \ldots, s_{n-1}, 0) & \text{if } v_n \notin \mathit{fanin}(v_i) \ , \\ f_{v_i}(s_1, \ldots, s_{n-1}, f_{v_n}(s_1, \ldots, s_{n-1}, 0)) & \text{if } v_n \in \mathit{fanin}(v_i) \ . \end{cases}$$

The above expression substitutes s_n by $f_{v_n}(s)$ and, therefore, $\widehat{f}_{v_i}(\widehat{s})$ does not depend on s_n, as $v_n \notin \mathit{fanin}(v_n)$. For this reason only combinational gates can be abstracted, because sequential gates depend on their own output.

Figure 4.15 shows how the boolean expression of a complex gate is derived from the expressions of the simple gates. In case $|\mathit{fanout}(v_n)| > 1$, multiple complex gates will be created, as illustrated in figure 4.16.



v_1
v_2
v_3
v_5
C
v_4
$f_4 = v_3 \cdot v_5 + v_4 \cdot (v_3 + v_5)$
$f_4 = v_3 \cdot (v_1 + v_2) + v_4 \cdot (v_1 + v_2 + v_3)$

Figure 4.15 OR and C gates collapsed into a complex gate.

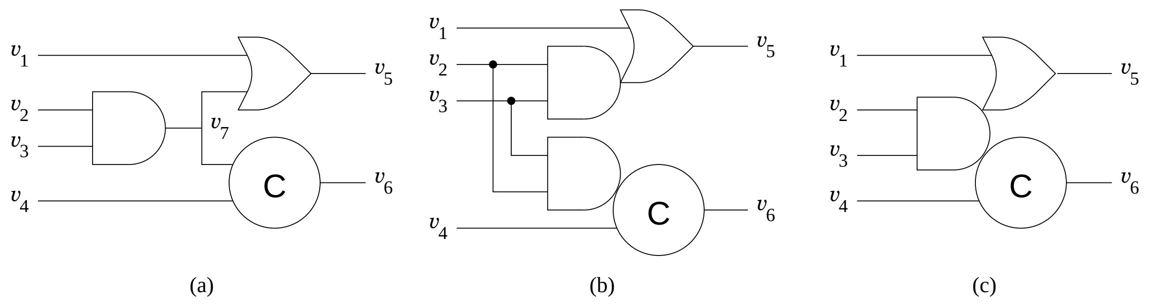


Figure 4.16 (a) Gate with multiple fanout. (b) Complex gate considered for functional correctness (c) and for behavioral correctness.

4.7.3 Projections adding nondeterminism

When a state graph is projected onto the interface signals of a complex gate, semantically different states might be mapped onto a same state. This could turn a deterministic sequence of events into a nondeterministic one. In other words, projection may not only hide some events, which is good, but also introduce inexistent traces that could cause a circuit malfunction.

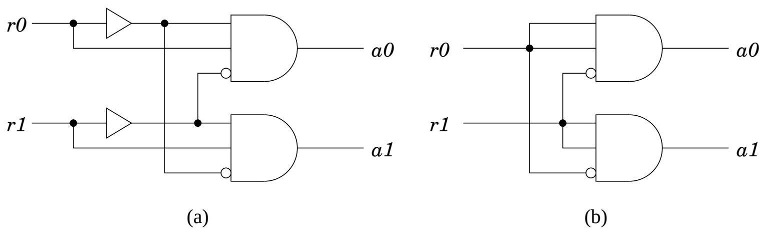


Figure 4.17 Example of (a) flat and (b) collapsed circuit.

As an example, we show the circuit in figure 4.17, provided by Peter Beerel and Jerry Burch [BB95], which work with alternating choice but not with nondeterministic choice. The original specification, in figure 4.18(a), has eight states, but there are two of them with the same code, $r0 = r1 = a0 = a1 = 0$. Thus, the projection onto $\{r0, r1, a0, a1\}$ gives the state graph in figure 4.18(b), with only seven states and nondeterministic choice. Under such an environment, the complex gate is not semi-modular. In one of the states with code 0000, the place $p0$ is marked, whereas in the other state it has no token. As figure 4.18(c) depicts, place $p0$ can be kept in order to disambiguate both semantically different states.

In practice, fortunately, introducing nondeterminism in the environment of a complex gate rarely causes circuit malfunction. Therefore, initially the state graph will be mapped only onto the interface of the complex gate. If a violation of semi-modularity occurs, we can always keep some variables alien to the complex gate and then rerun the second verification step. This procedure is

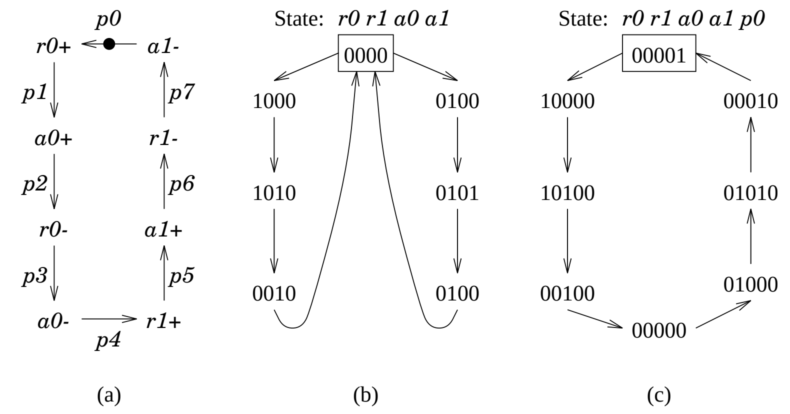


Figure 4.18 (a) Environment of circuit. Projections (b) adding nondeterminism (c) or not.

```
hierarchical_verification {
   $C'$ := collapse($C$);
   $SG'$ := obtain_SG($C'$,$N$);
   exit_if_$C'$_is_not_speed-independent;
   for each complex gate $G' \in C'$ {
      E := project($SG'$,$G'$);
      $G$ := original_gates($C$,$G'$);
      if (NOT is_semi-modular($G$,E)) {
         E := project_keeping_USC(SG,$G'$);
         if (NOT is_semi-modular($G$,E)) return FALSE;
      }
   }
   return TRUE;
}
```

Figure 4.19 Hierarchical verification algorithm

depicted in figure 4.19. The function `project_keeping_USC(SG,`G'`)` projects a state graph onto the set of variables of G' plus some additional variables such that all states have distinct codes.

4.7.4 Why is hierarchical verification exact?

In our framework, the absence of hazards is proved by verifying that the circuit is semi-modular, i.e. no gate can be disabled by changing the value of its inputs. Let us assume that C is a circuit and $\widehat{C}$ is an equivalent circuit in which some gates have been collapsed into complex gates and the corresponding internal signals eliminated. In this section we will prove that:

a) if $\widehat{C}$ is not semi-modular, then C is not semi-modular either.

b) if $\widehat{C}$ is semi-modular but C is not semi-modular, there is a complex gate of $\widehat{C}$ for which the behavior of the corresponding decomposed gate, under the projection of the state graph of $\widehat{C}$ onto the input/outputs of the gate, is not semi-modular.

Conjecture a) guarantees *no false negatives*, whereas conjecture b) guarantees no *false positives*.

We have considered a circuit to be a set of gates connected to an environment, modeled as a Signal Transition Graph. In particular, the behavior of the environment could also be modeled as a set of gates which inputs are connected to the outputs of the circuit and vice versa. The system resulting from such a connection, would not have interface, i.e. it would be an *autonomous circuit*.

Without loss of generality and for the sake of simplicity, we will consider autonomous circuits in subsequent definitions and proofs. The obtained results can be naturally extended to circuits with interface.

Given the state graph of an autonomous circuit, $SG(C) = \langle S, E, V\rangle$, relation E can be partitioned into n subsets as follows:

$$\begin{aligned} E_i &= \{(s,s') \in E | s_i = \overline{s'_i} \wedge \forall_{j \neq i} s_i = s'_i\} \ , \\ E &= \bigcup_{v_i \in \mathcal{V}} E_i \ . \end{aligned}$$

Note that the labeling function λ is the identity. This means that each state $s \in S$ is a bit-vector over $\mathcal{V}$ such that the ith element of s, denoted by s_i, specifies the value of signal v_i in state s.

Next, observational equivalence [Mil80] is defined. This is a concept that establishes an equivalence among those circuits that produce the same events on a given set of signals. For simplicity, we will use a restricted definition, since we are only interested in circuits in which the signals of one of them is a subset of the signals of the other.

Definition 4.3 (Observational equivalence between two circuits) *Let $C = \langle \mathcal{V}, \mathcal{F}, s^0\rangle$ and $\widehat{C} = \langle X, \widehat{\mathcal{F}}, \widehat{s}^0\rangle$ be two circuits, with $X \subseteq \mathcal{V}$, and let $SG(C) = \langle S, E, \mathcal{V}\rangle$ and $SG(\widehat{C}) = \langle \widehat{S}, \widehat{E}, X\rangle$ be their state graphs. C and $\widehat{C}$ are* observationally equivalent *iff:*

1. $\widehat{s}^0 = proj_X(s^0)$.
2. $\forall s \in S, \widehat{s} \in \widehat{S}$ *such that* $\widehat{s} = proj_X(s)$ *and* $\forall v_i \in X$*:*
   a) *if* sE_is' *then* $\exists \widehat{s}' \in \widehat{S}$ *such that* $\widehat{s}\widehat{E}_i\widehat{s}'$ *and* $\widehat{s}' = proj_X(s')$.
   b) *if* $\widehat{s}\widehat{E}_i\widehat{s}'$ *then* $\exists s' \in S$ *such that* $sE^*_{\overline{X}}E_iE^*_{\overline{X}}s'$ *and* $\widehat{s}' = proj_X(s')$.

   where $E^*_{\overline{X}}$ *denotes any sequence of non-observable transitions.*

In this section we propose to verify semi-modularity rather than speed-independence. As it has been explained in section 4.5.2, both concepts are tightly related for most practical cases.

Theorem 4.2 *Given two circuits $C = \langle \mathcal{V}, \mathcal{F}, s^0\rangle$ and $\widehat{C} = \langle X, \widehat{\mathcal{F}}, \widehat{s}^0\rangle$, with $X = \mathcal{V} - \{v_n\}$, $\widehat{\mathcal{F}}$ defined as above and $\widehat{s}^0 = proj_X(s^0)$, and their state graphs $SG(C) = \langle S, E, \mathcal{V}\rangle$ and $SG(\widehat{C}) = proj_X(SG(C)) = \langle \widehat{S}, \widehat{E}, X\rangle$, C and $\widehat{C}$ are observationally equivalent iff all signals in fanout(v_n) are semi-modular with respect to v_n in $SG(C)$.*

Proof
Condition 1 of definition 4.3 holds by construction. Let $s \in S$, $\widehat{s} \in \widehat{S}$, $\widehat{s} = proj_X(s)$ and $v_i \in X$. In those cases where we prove that $f_{v_i}(s) = \widehat{f}_{v_i}(\widehat{s})$, it immediately follows that observational equivalence holds. More precisely, $f_{v_i}(s) = \widehat{f}_{v_i}(\widehat{s}) = s_i$ implies that v_i is stable in both s and $\widehat{s}$ and, therefore, conditions 2.a and 2.b hold. If $f_{v_i}(s) = \widehat{f}_{v_i}(\widehat{s}) = \bar{s}_i$ there exists s' and $\widehat{s}'$ such that

sE_is' and $\widehat{s}\widehat{E}_i\widehat{s}'$ and $\widehat{s}' = proj_X(s')$, since the same signal transitions from s and $\widehat{s}$. Therefore, conditions 2.a and 2.b also hold.

If $v_n \notin fanin(v_i)$ then $\widehat{f}_{v_i}(\widehat{s}) = f_{v_i}(s)$ and, consequently, observational equivalence holds.

If $v_n \in fanin(v_i)$ then

$$\widehat{f}_{v_i}(\widehat{s}) = f_{v_i}(s_1, \ldots, s_{n-1}, 0) \cdot \overline{f_{v_n}(s)} + f_{v_i}(s_1, \ldots, s_{n-1}, 1) \cdot f_{v_n}(s) \ .$$

$\boxed{\text{semi-modularity} \Longrightarrow \text{observational equivalence}}$

Since v_i is semi-modular with respect to v_n, a change on signal v_n cannot disable v_i. Hence, $f_{v_i}(s)$ does not depend on s_n when signals v_i and v_n are simultaneously excited, i.e.

$$[f_{v_i}(s) \neq s_i \ \wedge \ f_{v_n}(s) \neq s_n] \quad \Longrightarrow \quad [f_{v_i}(s_1, \ldots, s_{n-1}, 0) = f_{v_i}(s_1, \ldots, s_{n-1}, 1)] \ .$$

For the previous predicate to hold we need

$$[f_{v_i}(s) = s_i] \ \vee \ [f_{v_n}(s) = s_n] \ \vee \ [f_{v_i}(s_1, \ldots, s_{n-1}, 0) = f_{v_i}(s_1, \ldots, s_{n-1}, 1)] \ .$$

If $f_{v_n}(s) = s_n$, we have (by Boole's expansion)

$$\widehat{f}_{v_i}(\widehat{s}) = f_{v_i}(s_1, \ldots, s_{n-1}, 0) \cdot \bar{s_n} + f_{v_i}(s_1, \ldots, s_{n-1}, 1) \cdot s_n = f_{v_i}(s) \ .$$

If $f_{v_i}(s_1, \ldots, s_{n-1}, 0) = f_{v_i}(s_1, \ldots, s_{n-1}, 1)$ it immediately follows that

$$\widehat{f}_{v_i}(\widehat{s}) = f_{v_i}(s_1, \ldots, s_{n-1}, 0) = f_{v_i}(s_1, \ldots, s_{n-1}, 1) = f_{v_i}(s) \ .$$

It only remains the case

$$[f_{v_i}(s) = s_i] \ \wedge \ [f_{v_n}(s) = \bar{s_n}] \ \wedge \ [f_{v_i}(s_1, \ldots, s_{n-1}, 0) = \overline{f_{v_i}(s_1, \ldots, s_{n-1}, 1)}] \ .$$

which describes the situation in which v_i is stable, v_n is excited, and $f_{v_i}(s)$ depends on the value of signal v_n. Hence,

$$\widehat{f}_{v_i}(\widehat{s}) = \overline{f_{v_i}(s_1, \ldots, s_{n-1}, 1)} \cdot s_n + f_{v_i}(s_1, \ldots, s_{n-1}, 1) \cdot \bar{s_n} \overline{f_{v_i}(s)} \ = \ \bar{s_i} \ .$$

Clearly, condition 2.a holds for state s, since v_i is not excited in s. To prove 2.b, let us take $\widehat{s}'$ such that $\widehat{s}\widehat{E}_i\widehat{s}'$. We will prove that there exists $s', s'' \in S$ such that $sE_ns''E_is'$ and $\widehat{s}' = proj_X(s')$.

Since v_n is excited in s then we have $s'' \in S$ such that sE_ns''. But now, v_i is also excited in s'' as

$$f_{v_i}(s_1, \ldots, s_{n-1}, 0) = \overline{f_{v_i}(s_1, \ldots, s_{n-1}, 1)} \ ,$$

and thus there exists $s' \in S$ such that $s''E_is'$. Finally, s and s' only differ in the ith and nth elements and therefore $\widehat{s}' = proj_X(s')$.

$\boxed{\neg \ \text{semi-modularity} \Longrightarrow \neg \ \text{observational equivalence}}$

If v_i is not semi-modular with respect to v_n, then $\exists s, s', s'' \in S$ such that sE_is', sE_ns'' and v_i is not excited in s''.
Since only v_n changes between s and s'', we have that $\widehat{s} = proj_X(s) = proj_X(s'')$. Thus,

$$\widehat{f}_{v_i}(\widehat{s}) = f_{v_i}(s_1, \ldots, s_{n-1}, 0) \cdot \overline{f_{v_n}(s)} + f_{v_i}(s_1, \ldots, s_{n-1}, 1) \cdot f_{v_n}(s) \ .$$

Since v_i is excited in s and stable in s'' (after a transition of v_n) then

$$f_{v_i}(s_1, \ldots, s_{n-1}, 0) = \overline{f_{v_i}(s_1, \ldots, s_{n-1}, 1)} \ .$$

Moreover, $f_{v_n}(s) = \bar{s_n}$ and $f_{v_i}(s) = \bar{s_i}$, as v_n and v_i are excited in s. Therefore,

$$\widehat{f}_{v_i}(\widehat{s}) = \overline{f_{v_i}(s_1, \ldots, s_{n-1}, 1)} \cdot s_n + f_{v_i}(s_1, \ldots, s_{n-1}, 1) \cdot \bar{s_n}\overline{f_{v_i}(s)} \ = \ s_i \ ,$$

which means that v_i is not excited in $\widehat{s}$ and, therefore, condition 2.a does not hold. □

Theorem 4.2 is the basis to prove that hierarchical verification is exact. This is the purpose of the next corollaries.

Corollary 4.1 *$\widehat{C}$ not semi-modular $\Longrightarrow$ C not semi-modular.*

Proof This immediately follows from the fact that the state graph of $\widehat{C}$ is the projection of the state graph of C. □

Corollary 4.1 guarantees that hierarchical verification will not give false negatives.

Corollary 4.2 *If $\widehat{C}$ is semi-modular and C is not semi-modular, then either v_n or some signal $v_i \in fanout(v_n)$ are not semi-modular in C.*

Proof (by contradiction) Assume that v_n and all its fanout signals are semi-modular. Then, by theorem 4.2, C and $\widehat{C}$ should be observationally equivalent. Since $\widehat{C}$ is semi-modular and C is not semi-modular, then the only non-observable signal, v_n, should be non-semi-modular, which contradicts the initial assumption. □

The following proposition is a result of definition 3.4 (projection of a state graph).

Proposition 4.1 *Let $C = \langle \mathcal{V}, \mathcal{F}, s^0 \rangle$ be a circuit, $SG(C) = \langle S, E, \mathcal{V} \rangle$ its state graph, and $fanin(v_i) \cup \{v_i\} \subseteq X \subseteq \mathcal{V}$. Let $proj_X(SG(C)) = \langle \widehat{S}, \widehat{E}, X \rangle$ be the projection of the original state graph onto X. Let $s, s' \in S$ and $\widehat{s} \in \widehat{S}$ such that $proj_X(s) = proj_X(s') = \widehat{s}$. Then*

$$v_i \textit{ excited in } s \Leftrightarrow v_i \textit{ excited in } s' \Leftrightarrow v_i \textit{ excited in } \widehat{s} \ .$$

Proposition 4.1 is crucial for our method, since it states that the excitation/stability of a complex gate, and subsequently semi-modularity, can be locally checked by only knowing the values of the input/output signals of the gate and regardless the state of the rest of the circuit.

Corollary 4.2 shows that hierarchical verification does not produce false positives. Consider a complex gate that drives $v_i \in fanout(v_n)$, and that X_i is the set of input/output signals of the gate, i.e.

$$X_i = \{v_i\} \cup (fanin(v_i) - \{v_n\}) \cup fanin(v_n) \ .$$

It can be derived that, by taking $proj_{X_i}(SG(\widehat{C}))$ as the environment of the complex gate, and s_n^0 as the initial value for signal v_n, non-semi-modularity of v_i and v_n in $SG(C)$ is also detected in $proj_{X_i}(SG(\widehat{C}))$ (by proposition 4.1). Intuitively, it can be proved by showing that there is always one state s of C in which non-semi-modularity is manifested for the first time from s^0. Because of the observational equivalence, while semi-modularity holds from s^0, the projection of s onto X_i will also belong to the set of states of $proj_{X_i}(SG(\widehat{C}))$.

4.7.5 Verification of isomorphic subcircuits

As figure 4.14 illustrates, a circuit may have different levels of hierarchy. In some of these levels isomorphic subcircuits may appear, for example the AND-OR gate in figure 4.14(c). A conservative way of speeding-up the second verification step that takes advantage of isomorphic subcircuits will be presented by means of an example.

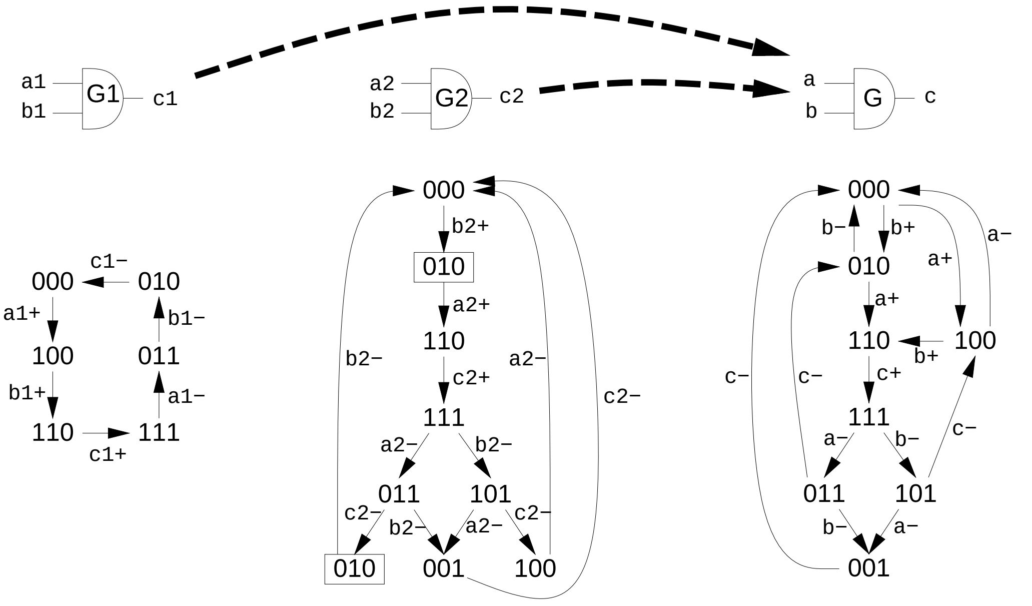


Figure 4.20 Union of environments for different instances of the same gate.

Let us assume that a circuit has several instances of a same complex gate. Figure 4.20 shows two AND gates of the same circuit with a different environment for each. In this example, AND gates are used for simplicity, since verification of semi-modularity in simple gates could have been done in the previous step. As previously mentioned, the environment of a complex gate is calculated as the projection of the state graph onto X_i. In this is example the states labeled with `010` in the environment of G2 result from the projection of two different states. For the sake of clearness, they are depicted as different states in the figure.

To verify the semi-modularity of each AND gate, we calculate the union of the environments of all AND gates of the circuit, shown as the environment for the generic gate G in figure 4.20. This many-to-one mapping may introduce choice and/or nondeterminism not manifested in the initial state graph. In fact, the set of sequences of transitions accepted by the union of projected state graphs can be larger than the union of the sets of sequences generated by each individual gate. However, semi-modularity is a local property of a gate that needs to be checked only between adjacent states of its environment (see proposition 4.1).

Interestingly, if the union of the projected state graphs produces a semi-modular behavior of the generic gate G, it also describes a set of sequences of events that, if applied to each gate individually, would produce a semi-modular behavior. In other words, if gate G is semi-modular with the union of environments, each gate will be semi-modular with its own environment.

The conservativeness of this approach may come from both the projection onto the interface signals, as explained in section 4.7.3, and the union of environments. The projection of the state graph onto the interface signals, as well as the union of environments, will keep the edges involving interface signal switches (see proposition 4.1), but they may also fold semantically different states onto the same state, thus introducing additional nondeterminism (choice). The whole approach is exact rather than conservative, since there is always the possibility of verifying each isomorphic subgraph separately.

Needless to say that, with the previous considerations, the presented approach allows to verify circuits against an environment, possibly containing choice, nondeterminism and/or state variables that do not correspond to values of input/output signals.

4.8 VERIFICATION OF PROPERTIES

Verification involves two different aspects: formally proving that a circuit does not behave incorrectly, i.e. conformance to the specification, and, moreover, that it behaves "well". At a first glance, it could seem that the former statement implies the latter and vice versa, but this is not necessarily true. Conformance to the specification only means that no unexpected behavior is observed at the outputs of the circuit. Let us think of a *block of wood* circuit (the *universal do-nothing module* proposed by Molnar). Such a circuit will accept any input transition but, obviously, will not produce any response at its outputs (shall we call them "outputs"). In conclusion, no failure state would be reached and the circuit would be reported as correct.

Thus, there is also a need of assuring that the system has certain properties, e.g. absence of deadlocks, liveness, home-state, or other more concrete properties like "all the markings in the environment are visited when connected to the circuit" or "the switching of signal x is acknowledged by switching signal y", etc.

In this section algorithms for proving *deadlock freeness* and the *home-state* property [Mur89] of the whole system, are presented.

Other properties can be verified by using different algorithms, but listing them all would make this section cumbersome. In particular, the algorithms in section 3.8 can also be used with minor changes in order to check if properties of the specification Petri net still hold when its behavior is restricted by the circuit.

4.8.1 Deadlock freeness

Figure 4.21 shows how deadlock freeness can be easily tested. A deadlock state is a state from which the system cannot make any progress. In a deadlock state neither any transition is enabled nor a gate is excited. States in which a transition t is not enabled are found by the formula $S \cap \overline{\mathrm{E}_t}$. Similarly, the product $S \cap \overline{f_v^+} \cap \overline{f_v^-}$ gives the subset of states in S in which signal v cannot switch. The characteristic function of the states that produce a deadlock is given by the product of the two previous formulae calculated for each transition and each gate.

4.8.2 Home state

The initial state s^0 of a system is a *home state* [Mur89] if s^0 can be reached from any other reachable state of the system. Apart from the interest of the property itself, the home-state property is a sufficient (but not necessary) condition for an L1-live system to be L4-live[1]. Clearly, if every

[1] *L4-liveness* and *home state* are concepts used for Petri nets that we naturally extend to circuits.

```
system_deadlock {
   Deadlock := Reached;
   for each transition $t \in \mathcal{T}_I$
      Deadlock := Deadlock $\cap$ $\overline{E_t}$;
   for each signal $v \in \mathcal{V}_G$
      Deadlock := Deadlock $\cap$ $\overline{f_v^+}$ $\cap$ $\overline{f_v^-}$;
   return Deadlock;
}
```

Figure 4.21 Algorithm for checking deadlock freeness.

transition is enabled in some reachable state (L1-liveness) and from any state the initial state s^0 can be reached, each transition can be fired infinitely often, i.e. the system is L4-live. Otherwise, L4-liveness can be verified by other techniques with higher complexity as it was explained in section 3.8.

The algorithm in figure 4.22 checks if the initial state s^0 is a home state. The state s^0 will be a home state if performing a backward traversal we reach the same states that going forward. Nevertheless, we restrict the states found backwards to the reachable set of states, because of the inherent nondeterminism when going backward. The algorithm is similar to a normal Breadth First Search, but at each step the new states are removed from the reachable set of states. The backward traversal completes when no more states can be removed. Only if `Removable` becomes the empty set, s^0 will be a home state.

```
home_state {
   Removable := Reached;
   From := $s^0$;
   do {
      New := $\emptyset$;
/* Let $\delta^{-1}$ be either $\delta_G^{-1}$, $\delta_I^{-1}$, $\delta_O^{-1}$ or $\delta_N^{-1}$ */
/* depending on whether $x$ in $\mathcal{V}_H$, $\mathcal{T}_I$, $\mathcal{T}_O$ or $\mathcal{T}_H$ */
      for each signal/transition $x \in \mathcal{V}_H \cup \mathcal{T}$ {
         New := New $\cup$ ($\delta^{-1}$(From,$x$) $\cap$ Removable);
      }
      From := New - Removable;
      Removable := Removable - From;
   } while (From $\neq \emptyset$);
   return (Removable = $\emptyset$);
}
```

Figure 4.22 Algorithm for checking the home-state property.

4.8.3 Verification of properties on the collapsed circuit

Let us assume $C = \langle \mathcal{V}, \mathcal{F}, s^0 \rangle$ to be a speed-independent circuit and $\widehat{C} = \langle X, \widehat{\mathcal{F}}, \widehat{s}^0 \rangle$ a complex-gate-equivalent circuit of C.

Corollary 4.3 *Properties concerning the observable behavior of C can be verified in $\widehat{C}$.*

Proof It directly follows from theorem 4.2, which states that a speed-independent circuit and its collapsed version are observationally equivalent. □

Consequently, if circuit $\widehat{C}$ reaches a deadlock state $\widehat{s}$, C will reach some state s, such that $\widehat{s} = proj_X(s)$, from which no change is produced in any signal $v \in X$. Note that this does not implies that the state s is a deadlock state, but a *live-lock* one. In other words, the only excited signals in s and in any s' reachable from s, if any, are signals inside the complex gates. Since by assumption collapsed signals are not observable, the environment will observe that the interface of the circuit is halted.

Similar reasoning can be done about other levels of liveness. Thus, if $\widehat{C}$ is L4-live, then the interface of C will show this same behavior, even though some group of collapsed signals are dead. Let us take figure 4.23 as an example. Clearly, both circuits have the same behavior, since the output of the OR gate is always set to one. Despite of having the same observable behavior, the circuit in figure 4.23(a) cannot be L4-live, because one of its signals is dead.

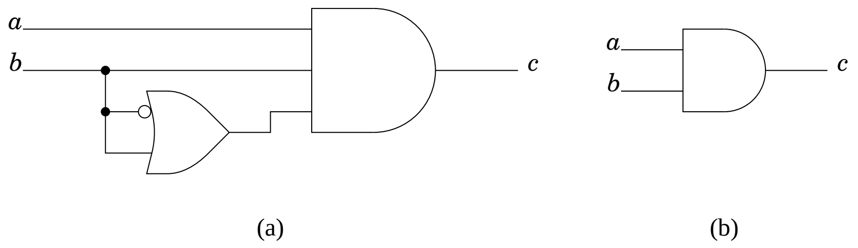


Figure 4.23 Two observationally equivalent circuits.

Corollary 4.3 also guarantees that properties concerning the causality between events in the circuit interface can also be verified in the collapsed circuit. For example if in circuit $\widehat{C}$ an event $x*$ is properly acknowledged by event $y*$, the original circuit C will generate this very same behavior.

4.9 RESULTS

Two tables illustrate the results obtained by running several experiments on our verifier. Table 4.1 reports the outcome of verifying speed-independent circuits, whereas table 4.2 presents the results achieved when trying to verify non-speed-independent circuits. The end of this section is devoted to compare our results to those given by other authors.

Verification of speed-independent circuits

The first example in table 4.1 is a circuit automatically generated following the techniques presented in [Pas96]. The other circuits are scalable, i.e. they can be enlarged by simply increasing the number of instances of the basic cells. However, their intrinsic regularity has intentionally not been exploited to verify the circuit. The scalable circuits are the following: a Distributed Mutual Exclusion (DME) arbitration circuit [Mar85, Dil89], a tree arbiter [Sei80b], an asynchronous FIFO [Mar86], a register file [NUK$^+$94] and a demultiplexer [BM92].

Results on flat and hierarchical verification are shown. The signals for hierarchical verification are output or state signals, since internal combinational gates are eliminated. The number of states of hierarchical verification is the one obtained during functional verification. The reported BDD sizes are the largest ones encountered during the traversal of the circuit. The number of

example	signals		states		BDD size		iterations		CPU (sec.)		speed
	flat	hier.	flat	hier.	flat	hier.	flat	hier.	flat	hier.	up
master-read	27	14	1.7×10^5	2.1×10^3	6432	1091	13	13	42	5	8.4
4 DME	72	36	7.5×10^4	4.1×10^3	1963	1082	34	19	39	7	5.6
8 DME	144	72	8.0×10^8	2.1×10^6	5175	3035	58	31	428	86	5.0
16 DME	288	144	4.5×10^{16}	2.8×10^{11}	11599	7181	106	55	3890	799	4.9
32 DME	576	288	7.0×10^{31}	2.4×10^{21}	24751	16391	202	103	28365	5788	4.9
6 Tree arb.	42	28	2.3×10^5	6.3×10^4	3253	2457	22	22	40	25	1.6
8 Tree arb.	58	38	1.4×10^7	2.3×10^6	9084	6239	30	27	261	132	2.0
10 Tree arb.	74	48	7.9×10^8	8.4×10^7	20789	9184	39	37	1425	512	2.8
12 Tree arb.	90	58	4.5×10^{10}	3.0×10^9	26717	14703	49	45	4316	1318	3.3
5 FIFO	18	–	2.7×10^3	–	573	–	12	–	8	–	–
10 FIFO	33	–	1.2×10^6	–	1765	–	22	–	130	–	–
15 FIFO	48	–	5.8×10^8	–	3662	–	32	–	634	–	–
20 FIFO	63	–	2.7×10^{11}	–	6214	–	42	–	2200	–	–
4-bit REG	74	50	7.6×10^7	1.2×10^6	14000	10412	15	7	388	171	2.3
6-bit REG	109	73	2.3×10^{11}	4.6×10^8	40346	31008	15	7	2156	681	3.2
8-bit REG	144	96	7.1×10^{14}	1.7×10^{11}	93913	70133	15	7	7246	1763	4.1
10-bit REG	179	119	2.2×10^{18}	6.7×10^{13}	188040	149250	15	7	11330	4650	2.4
6-bit DMUX	91	49	1.6×10^{17}	8.2×10^9	19057	3083	9	5	662	95	7.0
8-bit DMUX	121	65	6.9×10^{22}	1.3×10^{13}	26473	4117	9	5	1268	181	7.0
10-bit DMUX	151	81	2.9×10^{28}	2.1×10^{16}	33889	5151	9	5	2027	292	6.9
12-bit DMUX	181	97	1.3×10^{34}	3.4×10^{19}	41305	6185	9	5	2995	444	6.7

Table 4.1 Experimental results with speed-independent circuits.

iterations of the verification algorithm is also reported. CPU times have been obtained on a Sun SPARCStation-20 with 128 MB memory.

All the circuits are dominated by memory elements. The one with most combinational gates is the DME (half of the signals). The FIFO is a peculiar case, as all the signals are outputs of memory elements and, therefore, no difference exists between flat and hierarchical verification. The execution time of hierarchical verification is mostly dominated by the first step (functional verification).

The size of the BDDs, often crucial to avoid running out of memory space, is reduced by the fact that many variables are eliminated when reducing to complex gates. The significant improvements in CPU time are basically due to two factors: 1) the reduction of the size of the BDDs and 2) the reduction of the logic depth of the circuit, which directly influences on the number of iterations required to reach a fixed point during the traversal.

The presented results confirm that hierarchical verification makes depend time complexity on the number of state signals of the circuit, rather than on the number of gates. We believe that even better results can be obtained for circuits generated by automatic synthesis techniques, in which the ratio of combinational gates may be higher. However, at this moment there are no examples large enough to be considered critical for verification (the largest ones can be verified in roughly a dozen of seconds). As tools for synthesis and composition of circuits become more mature, the complexity of the circuits will significantly increase.

Verification of non-speed-independent circuits

There are several synthesis methodologies that do not assume unbounded delays for their elements [Van90, Lav92]. Consequently, the synthesized circuits will not necessarily be speed-independent. The examples in table 4.2 are included to show how our verifier is also able to detect

example	signals	iterations	CPU (sec.)
atod	22	1	0.25
chu150	13	3	0.17
converta	21	5	0.34
nowick	14	6	0.24
rpdft	18	3	0.32
vbe6a	41	2	0.46

Table 4.2 Experimental results with circuits not satisfying its specification.

when a circuit does not satisfy its specification. The circuits have been generated from STG specifications by using SIS [SSL+92]. We want to remark that the circuits generated by SIS are correct if a bounded gate delay model is assumed and that those circuits have only been chosen to illustrate some non-speed-independent circuits.

Comparing results with other authors

The flat verification algorithm will be contrasted with the work by Burch et al. In [BCL+94] an extensive discussion on the DME example is presented. They build the reachability set of the circuit and then use temporal logic to check different properties. In particular they verify that never two or more requests are simultaneously granted.

Burch et al. model the asynchronous circuit by using the *partitioned transition relation*, and present different traversal algorithms. Their best results are obtained by using the *disjunctive* partitioned transition relation and the algorithm they call *Modified Breadth First Search (MBFS)*. In order to accelerate the reachability analysis, the authors also propose starting the BFS algorithm from several initial states.

We have used a similar variable ordering, the same set of initial states and we have run these benchmarks on a SPARC +1 machine, thus allowing a fair comparison. In both their and our approach CPU time is mainly dominated by reachability analysis. Our CPU times are comparable to those obtained in [BCL+94], as it can be seen in figure 4.24. The corresponding lines are respectively labeled "Flat" and "Burch et al."

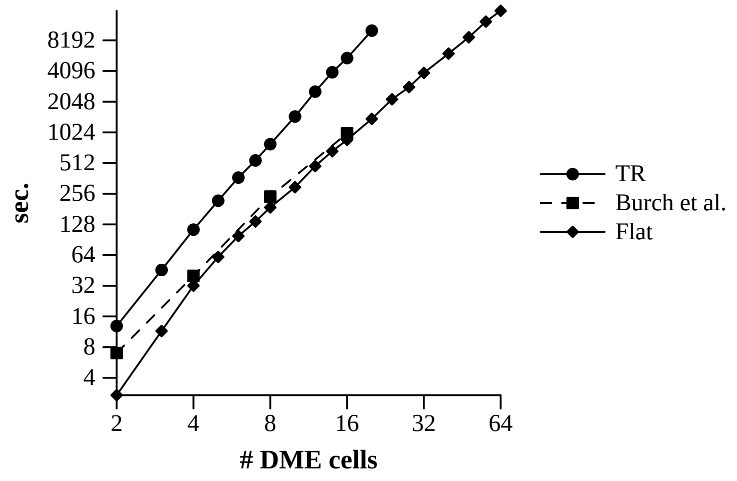


Figure 4.24 Comparison with Burch et al.

It is worth mentioning that in [BCL$^+$94] a state graph is used as model for the specification, whereas we use a Petri net. Therefore, the number of variables used in the symbolic traversal algorithms is smaller in their approach than in ours. Despite this disadvantage, our results are still comparable.

In [BCL$^+$94] they claim that transition function methods are less efficient that transition relation methods. However, by using the improvements presented in section 3.7 we have achieved a very fast symbolic traversal algorithm. In order to illustrate the advantage of our method we have verified several DME arbiters against their Petri net specification using our traversal algorithm and the Modified Breadth First Search algorithm proposed in [BCL$^+$94]. The outcome of this comparison is also depicted in figure 4.24, marked with the label "TR". If we used the transition relation our method would be about one order of magnitude slower.

Recall that in the comparisons above we have used the flat verification, since we wanted to show that our first approach was at least as good as the techniques proposed by other authors. This gives relevance to the gain achieved by the hierarchical verification method, which has only been compared to the flat one.

4.10 CONCLUSIONS

This chapter has presented how to verify speed-independent circuits against a Petri net (STG) specification or environment. This Petri net may specify input/output choice and non-determinism. Verification is performed by composing the circuit and its environment, thus forming a closed system. The reachability analysis of this system allows to detect any reachable state that violates the specification. In order to help to find design errors, diagnosis of erroneous circuits is also provided, in terms of a feasible trace leading from the initial state to a state violating the specification. Moreover, different properties can be verified at both levels, environment and circuit.

Since the closed system formed by the circuit and its specification is modeled by using boolean functions, efficient symbolic BDD techniques can be used for verification.

The complexity of formal verification of asynchronous circuits fundamentally depends on the size of the circuit, i.e. the number of gates. Reducing the size of a circuit by collapsing gates into complex gates only allows a partial verification in which a false positive might be given as a result. However, the necessary and sufficient conditions assuring that a circuit and its complex-gate-equivalent version are observationally equivalent have been proven in this chapter. Consequently, we have proposed a two-step algorithm that takes advantage of the circuit complexity reduction by using complex gates, but assures exact verification of speed-independence.

A verifier based on symbolic model checking has been implemented and several experiments with large circuits reported. It has been shown that, by reducing the number of relevant variables during verification, both the size of the BDDs and the computational cost drastically drop.

As future work, techniques for automatic regularity extraction will be explored [RK93]. They should allow to further reduce the computational cost of those circuits in which combinational gates dominate over memory elements.

5 TEST PATTERN GENERATION FOR ASYNCHRONOUS CIRCUITS

If the aborigine drafted an IQ test, all of Western civilization would presumably flunk it.
– Stanley Garn.

About this chapter

This chapter is devoted to the particular aspects of testing of asynchronous circuits. The need for testing has been previously justified in section 1.1.3. This chapter presents a novel approach to test asynchronous circuits by using synchronous test patterns.

5.1 INTRODUCTION

Testing is one of the crucial problems that remains to be satisfactorily solved for asynchronous circuits. Since they are implemented as arbitrary interconnections of gates and their behavior is not subordinated to the timing dictated by a global clock, controllability and observability of internal signals become significantly more costly than in synchronous circuits. Asynchronous circuits tend to have more feedbacks and more state holding elements than their synchronous counterparts. Thus, test pattern generation is harder and *design for testability* techniques like full *scan-path* may be unacceptably expensive. Furthermore, testers are inherently synchronous and cannot properly reproduce the environmental behavior for which an asynchronous circuit is designed.

5.1.1 Terminology

A *fault* is any defect that can cause a circuit malfunction. Faults are further studied in section 5.2. An *input vector* or *input pattern* is a set of values applied to the inputs of a circuit. Usually it is simply referred as a *vector* or *pattern*. A *test* is a sequence of input patterns.

Definition 5.1 (controllability) *A test makes a fault* controllable *if it produces a discrepancy in the node supposed to be faulty.*

Definition 5.2 (observability) *A test makes a fault* observable *if it propagates a discrepancy to the observable outputs.*

Definition 5.3 (strong detectability) *A test is said to* strongly detect *a fault if when applied to the circuit the fault is always detected.*

For simplicity, by fault detection we will always mean strong detection.

5.1.2 Previous work

Several studies have been presented in the last years addressing the testing and design for testability of asynchronous circuits.

Some classes of asynchronous circuits are *self-checking* [MH91, BM92, DGY95] for stuck-at faults, i.e. in presence of a stuck-at fault a circuit will halt while it is being normally operated, a situation easily detectable. Different fault models must be considered depending on the class of asynchronous circuit:

- Since delay-insensitive circuits work independently of the delays of gates and wires in the circuit, every transition in the circuit must be acknowledged by the receiver of such transition [MH91]. This fact makes that any stuck-at fault will cause the circuit to halt because such protocol will not be completed. Thus, delay-insensitive circuits are self-checking under the input stuck-at fault model.
- Speed-independent circuits assume negligible wire delays, hence a transition on the input of a fork needs only to be acknowledged by one of the recipients. This condition is equivalent to assuming all forks to be *isochronic* [Mar90a], i.e. a transition on the input of a fork arrives simultaneously at the outputs of the fork. Therefore, a transition on the input of the fork has to be acknowledged by at least one of the recipients and thus, a speed-independent circuit is self-checking under the output stuck-at fault model [BM92].
- Finally, quasi-delay-insensitive circuits are built of basic elements that have delay-insensitive interfaces, and only use isochronic forks inside such elements. For this type of circuits, the *isochronic transition fault model* is introduced [RS93], which considers input stuck-at faults for non-isochronic forks and output stuck-at faults for isochronic forks. Under such model, every quasi-delay-insensitive circuit is self-checking.

The main drawback of self-checking testing is that covering all faults may be impractically long. In addition, not all asynchronous circuits are self-checking due, for example, to the presence of redundancies to avoid hazards. These reasons have encouraged the use of techniques to increase the testability of asynchronous circuits. A simple design-for-testability approach can be based on the addition of test points into the circuit: *observation points* used to access an internal node by making it a primary output, and/or *control points* used to set the value of an internal node from a primary input. Optimal insertion of test points is a difficult task [Haz92] and usually expensive in terms of I/O pins. However, the number of them used for test points can be reduced at the cost of added test time, by storing the value of the test points in an internal register whose contents can be shifted in and out serially. This scheme requires only a few extra I/O pins regardless of the number of test points. Generalizing this idea leads to the introduction of a so called *scan-path*, where the registers in the circuit are extended to be *scan-registers*. In normal operation they work exactly as a register, but in test mode they form a shift-register as the scan-output of one register is connected to the scan-input of the next. Their content can be serially shifted-in to achieve full controllability, and shifted-out to achieve full observability. By transforming all the registers in a circuit to be scan-registers, the circuit is divided into a scan-path with combinational logic blocks in between. Thus the problem of generating test vectors is reduced to such combinational blocks.

The cost of so simpler test generation is an area increase and a potentially longer test time because test vectors must be serially shifted in and out.

Several works have been presented based on these ideas [LKL94, KLSV95, BCR96]. Other like [KB95] uses partial scan on some specific registers. Or as in [RS93] where instead of adding a scan-path to the circuit, a single test signal is added. By introducing the test signal and modifying some of the basic building blocks, a circuit can be tested in linear time regarding its size. Given that the circuit is self-checking under the isochronic transition fault model, it is tested by executing a single computation in test mode. The circuit is fault-free if a full handshake is performed on a particular channel. The cost of this approach is an important area increase due to the extra logic added to the basic building blocks.

Designing circuits under absolute delay assumptions, can lead presumably to smaller and faster circuits. However, testing these circuits must also check the assumed delay properties. Under the *path delay fault model* [Smi85], a given path in the fabricated circuit is faulty if it has a delay outside the specified interval. To test a circuit under this model, the delay of all paths in the circuit must be determined. The delay of a given path π is tested by applying two vectors $\langle V_1, V_2 \rangle$ at times t_0 and t_1, respectively. The time between t_0 and t_1 is long enough to assure that all nodes are stable at time t_1. The test vector has the property that when V_2 is applied after V_1, it causes a transition on all nodes along π. If the output of the circuit, latched at time t_2, differs from the specification, the delay on π is larger than $t_2 - t_1$ and a path delay fault is detected. A circuit can be made path delay fault testable by changing all state holding elements to scan elements [LKL94, KLSV95]. However, in contrast to the scan elements used in the scan-paths, they must be able to hold two values corresponding to a bit in the two vectors $\langle V_1, V_2 \rangle$.

A *robust* test is a test for a path delay fault which is independent of delays in gates, but not on the path under test [LR87]. There may exist paths in a circuit that are not robust path delay fault testable. In [KLSV91], a technique to make any boolean function delay fault testable is presented.

An alternative approach to make all paths delay fault testable is *variable phase splitting* [LKL94]. The idea is to make both phases of the variables controllable. If both a variable and its negated are used in a block of combinational logic, the variable is split into two independent variables. Under normal operation one will be the inverted of the other, but they will be independently controllable under test. Variables are stored in registers which have outputs for the true and the negated version of the variable, and must be able to hold two values in order to be able to perform the delay test.

Testing circuits under the path delay fault model is much more expensive than other approaches. The scan-registers must hold two bits, test points are required, and at least twice as many vectors are needed. Furthermore, the skew on the clock to the scan-registers must be accurately controlled.

Since commercial testers are inherently synchronous, some authors have proposed testing asynchronous circuits by synchronous test vectors. In [Bre74, BCR96], feedback loops are cut by virtual synchronous flip-flops during ATPG. Thus, ATPG can be done by using standard state-of-the-art synchronous techniques, but the obtained test vectors must be afterwards validated on the asynchronous circuit.

5.1.3 Overview

Since the available commercial testers are synchronous, the presented approach proposes using synchronous test vectors to test asynchronous circuits. The overview of such a methodology is as follows:

1. The non-faulty circuit is analyzed to find all input sequences that can be used in *fundamental mode with multiple input changes*, i.e. "a la synchronous". Any sequence producing either non-confluence or oscillation is discarded. After this analysis, the asynchronous circuit is modeled as a synchronous state machine (the *Confluent Stable State Graph*, *CSSG*, presented in section 5.4) with stable and deterministic behavior.

2. A symbolic ATPG strategy on the *CSSG* looks for a test sequence for each uncovered fault. Tests are found by a three-phase ATPG algorithm: first the states exciting a fault are searched, then those states are justified from the initial state or states and last the discrepancy is propagated to some output signal.

3. In order to accelerate the ATPG algorithm, two well known techniques are used:
   (a) *Random Test Pattern Generation* (Random TPG) on the *CSSG* is initially used to quickly cover a significant number of faults.
   (b) *Fault simulation* of each obtained test sequence is tried on every remaining faulty circuit to find all other faults covered by the same sequence.

Previous approaches that also proposed testing asynchronous circuits by synchronous test vectors [Bre74, BCR96] use a synchronous model of the asynchronous circuit. Thus they have to validate the obtained test vectors a posteriori on a more accurate model of the asynchronous circuit. On the contrary, we analyze the asynchronous circuit to find out those vectors that can be used in ATPG, so no further validation is needed. In addition test vectors generated with our methodology are independent from the technology and the gate delays. Further discussion can be found in section 5.7.1.

The results are presented in section 5.7. Speed-independent circuits are 100% output stuck-at testable in normal operation mode, and we show that with our method this fault coverage is maintained. In addition we are able to cover more than 99% input stuck-at faults. Asynchronous hazard-free with bounded delay circuits also showed a high fault coverage, although some particularly pathological benchmarks appeared.

5.2 FAULTS AND FAULT MODELS

Faults in a circuit may appear because of multiple failure mechanisms: a short-circuit between two or more lines (or simply a *short*), a broken line causing an *open*, defects causing wrong transistor threshold voltages or an increase in the resistivity of some line, etc. Despite of faults being physical mechanisms, most testing techniques deal with logical faults rather than with physical ones. Logical fault models are less accurate than physical fault models, but the former have been traditionally much more popular because of the following reasons:

1. Logical faults provide an abstraction of physical faults by mapping malfunction caused by several physical faults into a same logical fault. Therefore, testing becomes a logical problem rather than a physical one.

2. Most logical fault models are technology independent. Since no physical implementation of the circuit is needed to perform logical test, circuit testability can be checked (and, if needed, improved by using design for testability techniques) as soon as a circuit netlist is obtained.

3. Experience shows that tests that detect logical faults also cover many other difficult-to-model faults.

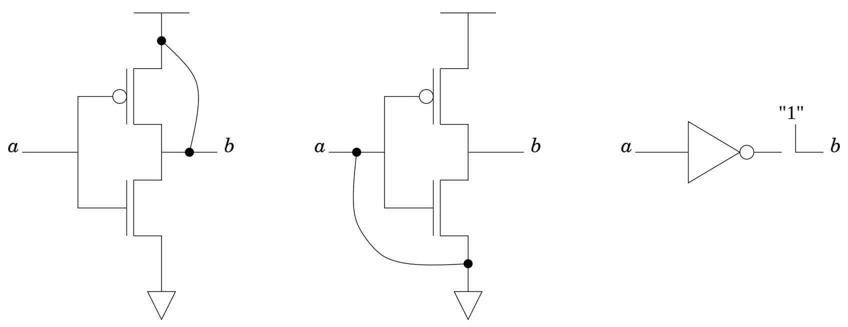


Figure 5.1 Two different *short* faults mapping to the same stuck-at 1 fault.

The most widely used logical fault model is the *stuck-at* model. Gates are assumed to be always correct and the only faults are possible in the circuit wires. A fault is modeled as an open line and a short-circuit between the line and the logical value zero or one (see figure 5.1). Depending on the location of the stuck-at value, two models are possible: the *output stuck-at* and the *input stuck-at* models. In the former faults can only appear at gate outputs, whereas in the latter inputs and output of the gate are possible fault sites. The difference between both models appears when a line has more than one fanout gate. Depending on the model, either a single fault location or $n+1$ fault sites are considered (being n the number of fanout gates). Figure 5.2 depicts the differences between both models.

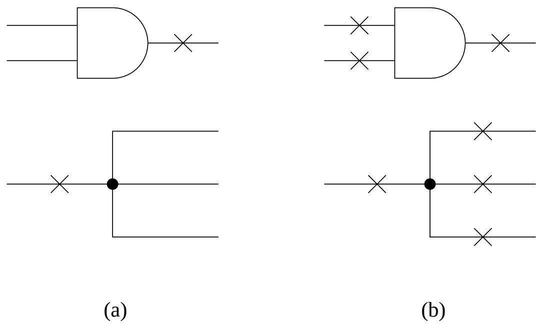


Figure 5.2 Fault location in the output (a) and input (b) stuck-at models.

Given a circuit with n signals, the number of faults under the output stuck-at model is clearly $2n$. If the input stuck-at model is used, this number is

$$2n + 2\sum_{i=1}^{n} k_i, \text{ where } k_i = \begin{cases} 0 & \text{if signal } i \text{ unconnected} \\ \mathit{fanout}(\text{signal } i) - 1 & \text{otherwise} \end{cases} .$$

In principle a test for every different fault should be searched, but, fortunately, many of those faults cause the same effect. Two faults are called *equivalent* if the function of the gate in presence of both faults is the same. Therefore, there is only need to look for a test for each equivalence class fault. Figure 5.3 shows examples of equivalent faults. In figure 5.3(a) three different faults, a stuck-at-1, b stuck-at-1 and c stuck-at-0, cause the function of the gate to be $f(a,b,c) = 0$. Figure 5.3(b) illustrates how either a stuck-at-0 or b stuck-at-0 make the gate to compute the function $f(a,b,c,d,e) = cd$.

From now on we will assume that at most one fault appears in the circuit. Experience shows that in most cases a multiple fault is detected by applying the tests derived for each one of its component faults.

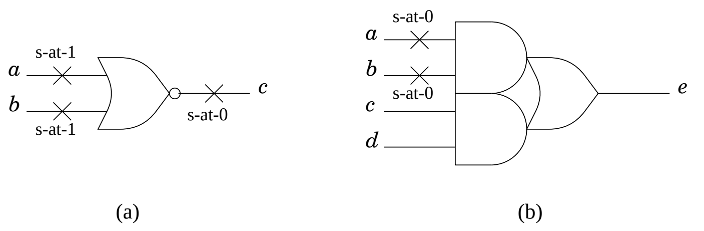


Figure 5.3 Examples of equivalent faults in NOR (a) and AND-OR (b) gates.

5.3 ASYNCHRONOUS CIRCUIT-UNDER-TEST BEHAVIOR

Given a circuit under test, we assume $\mathcal{V}_I$ to be the set of controllable inputs, whereas $\mathcal{V}_O$ is the set of observable outputs. A synchronous test procedure consists in setting the signals in $\mathcal{V}_I$ to some particular values or *input pattern*, wait a certain time for the circuit to stabilize and then observe the outputs. As figure 5.4 depicts, the same patterns are applied to both correct and faulty circuits. When the values at the outputs of the faulty circuit differ from the expected values, the fault is detected.

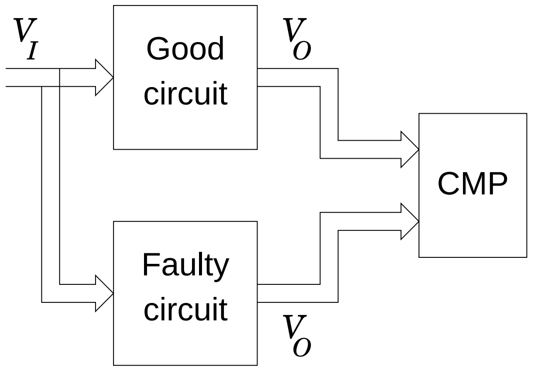


Figure 5.4 Goal of testing: observing the discrepancies between good and faulty circuits when the same patterns are applied.

Unlike synchronous circuits, asynchronous circuits may manifest non-deterministic and/or unstable behavior if an inappropriate environment is applied to their inputs. This is a consequence of a usually contrived circuit topology, where many gates have reconvergent fanout. Two problems may arise if input patterns are not selected conveniently: *non-confluence of settling state* and *oscillation*. Assuming a logic model for the circuit, these are the only source that may invalidate using a test vector. The former occurs when the final stable state of the circuit can be different depending on the arrival times of the input events and the delays of the internal gates. This phenomenon can potentially lead to metastability [Sei80a]. The latter occurs when the circuit cannot rest in a stable state.

To show non-confluence of settling state we will take the circuit in figure 5.5(a). Let us assume the circuit is in the stable state $ABabcdey = 01010000$ and that the input pattern $AB = 10$ is applied. Even if A and B change simultaneously, because of the delay on primary inputs due to input pads, we cannot assure that both a and b change at the same time. A "competition" between all sensitized paths starts as soon as some input is switched. Figure 5.6 shows the state graph generated by applying such pattern to the circuit in figure 5.5(a). Stable states are represented as

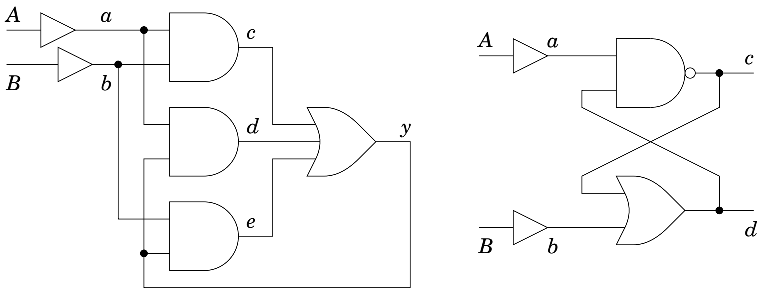


Figure 5.5 Circuits showing (a) non-confluence of settling state and (b) oscillation.

boxes, whereas unstable ones are in ellipses. It is not difficult to see that if gate c is slow to fall the stable state 10101101 will be reached, otherwise the circuit will settle to state 10100000.

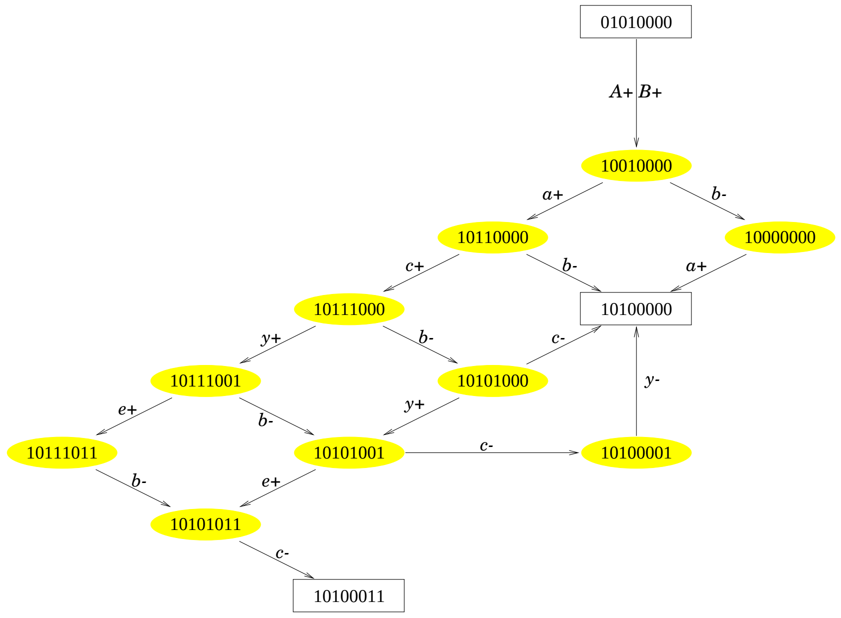


Figure 5.6 State graph of a circuit showing non-confluence.

The other problem is oscillation or cycles of unstable states. Let the circuit in figure 5.5(b) be in the initial stable state $ABabcd = 000011$. If input A is set to 1, the circuit starts oscillating. After $a+$, the sequence of transitions $c-, d-, c+, d+$ is repeatedly generated and the circuit never stabilizes. Other circuits may present transient oscillations that should or should not be avoided depending on the maximum desired settle time.

Thus, there is a need of providing some technique that assures using only *valid* test vectors, i.e. input patterns that produce neither non-confluence of the settling state nor indefinite or too long oscillation cycles[1].

[1]In section 5.4.1 this notion of "too long" will be discussed.

5.3.1 Circuit model

The model assumed for asynchronous circuits is analogous to the one described in section 4.3. Asynchronous circuits are modeled following the *unbounded gate delay* model [MB59]. Under such a model, delay elements are attached only to gate outputs and the delay magnitude is positive and finite, but unknown. The delay type we assume is *inertial* delay, i.e. pulses shorter than the delay magnitude are filtered out. The pair formed by a gate and its associated delay is referred simply as a *gate*.

Test vectors that would be valid under a bounded delay model, might be considered invalid under an unbounded delay model. On the other hand, test vectors generated assuming unbounded delays will also work on circuits with bounded delays. Therefore, our methodology while pessimistic, is independent of those aspects that may vary the gate delay, such as the technology, the fabrication process or the temperature at which the chips are being tested.

Each *primary input* of a circuit will be modeled as the input of a gate implementing the identity function. The circuits in figure 5.5 illustrate how primary inputs (A and B) are modeled. These buffers introduce the idea of delay associated to primary inputs.

5.3.2 Circuit State Graph (CSG)

In synchronous circuits the state depends on a subset of circuit signals called *state signals*. Usually this subset includes input and flip-flop signals. The order of the transitions along combinational paths is not relevant. The only limitation is that they all must occur in a limited *cycle time*. On the contrary, asynchronous circuits often have a more complicated structure. Since feedback loops are not cut by clocked flip-flops, the state of an asynchronous circuit is defined by all the binary values of both primary inputs and gates, rather than on a small subset of them.

Some of the following concepts may be slightly different from what was intended for verification. The definition of the state graph of a circuit and the next state function of a gate are modified in order to ease the mathematical formulation of testing.

A *circuit state graph* (CSG) is a 7-tuple $\langle \mathcal{S}, \mathcal{E}, \mathcal{P}, \mathcal{G}, S^0, \lambda_P, \lambda_G \rangle$, where $\mathcal{S}$ is a set of states, $\mathcal{E} \in \mathcal{S} \times \mathcal{S}$ is a set of edges or transitions between states, $\mathcal{P} = \{p_1, \ldots, p_m\}$ is the set of primary inputs, $\mathcal{G} = \{g_1, \ldots, g_n\}$ is the set of gates, and S^0 is the set of initial states. The labeling functions $\lambda_P : \mathcal{S} \longrightarrow \mathrm{B}^{\mathrm{m}}$ and $\lambda_G : \mathcal{S} \longrightarrow \mathrm{B}^{\mathrm{n}}$ map each state s with a binary vector consisting of the values in s of, respectively, primary inputs and gates. Regarding at the circuit in figure 5.5(a), $\mathcal{P} = \{A, B\}$ and $\mathcal{G} = \{a, b, c, d, e, y\}$. A state s in that circuit is given as a vertex of B^8, in particular, $s = ABabcdey$. The labeling functions "take" the part of the state corresponding to the inputs and the gates, i.e. $\lambda_P(s) = AB \in \mathrm{B}^2$ and $\lambda_G(s) = abcdey \in \mathrm{B}^6$.

Under the unbounded gate delay model the next state of a circuit uniquely depends on its present state. A gate is said to be *excited* if its output differs from the function it implements, and *stable* otherwise. If all the gates in a circuit are stable, the circuit is in a *stable state*. A *next state function* $\delta : \mathcal{S} \times \mathcal{G} \longrightarrow \mathcal{S}$ can be defined for each gate. Function $\delta(s, g_i)$ returns either the state reached by switching the output of g_i if it is excited or s if g_i is stable.

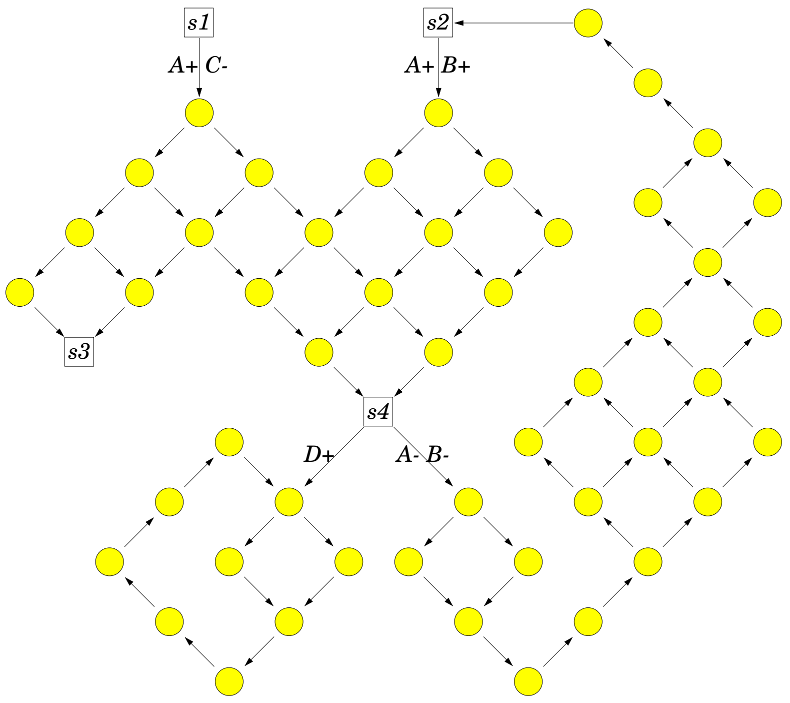
By using the next state function of each gate, the transition relation associated to circuit gates can be defined as

$$R_\delta = \{(s, s') \in \mathcal{S} \times \mathcal{S} \mid (s \text{ is stable } \wedge \ s = s') \ \vee \ (\exists g_i \in \mathcal{G} \text{ such that } s' = \delta(s, g_i) \neq s)\} \ .$$

For each pair $(s, s') \in R_\delta$, if s is stable, its successor is the same s, otherwise the successor is obtained by switching an excited gate.

5.3.3 *CSG* in test mode (*TCSG*)

In our approach, asynchronous circuits are tested in synchronous mode: provided the circuit is stable, an input vector is applied and the circuit is allowed to, eventually, settle. The time between the application of two input patterns is called the *test cycle*. Until otherwise noted, we will assume the test cycle is *long enough* to let the circuit stabilize (unless it oscillates). Figure 5.7(a) illustrates a possible *CSG* in test mode. In principle, in a circuit with n inputs, the number of possible input patterns is 2^n, but in this picture only a few patterns are represented for the sake of simplicity. Labeled boxes represent stable states, while shaded circles are unstable states. The outgoing arcs from a stable state are labeled with the changes at the circuit primary inputs. Only in such arcs it is allowed more than one signal transition, whereas outgoing arcs from unstable states represent single signal transitions. The latter are not labeled for clarity. We will refer to this circuit state graph in test mode as *TCSG*.



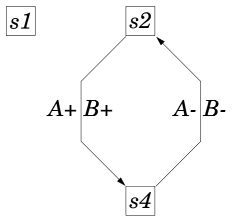


Figure 5.7 (a) A *TCSG* and (b) its corresponding *CSSG*.

The transition relation associated to input signals can therefore be defined as follows:

$$R_I = \{(s, s') \in \mathcal{S} \times \mathcal{S} \mid s \text{ is stable } \wedge \ \lambda_P(s) \neq \lambda_P(s') \ \wedge \ \lambda_G(s) = \lambda_G(s')\} \ .$$

Relation R_I describes all input patterns that can be applied to a stable state. Thus, sR_Is' if s is stable and s' only differs in certain number of inputs. This represents the situation in which several inputs have been changed but no gate has begun to switch yet.

The transition relation of a circuit in test mode is defined as $R = R_I \cup R_\delta$. Consequently, we can formally define a *TCSG* as a *CSG* such that $\mathcal{S}$ and $\mathcal{E}$ are strictly defined by the following recursion:

1. $S^0 \subseteq \mathcal{S}$.
2. $s \in \mathcal{S} \wedge sRs' \Rightarrow \begin{cases} s' \in \mathcal{S} \\ (s, s') \in \mathcal{E} \end{cases}$

The set $\mathcal{S}$ is the set of reachable states of a circuit in test mode, while $\mathcal{E}$ is the transition relation R restricted to $\mathcal{S}$. $\mathcal{S}$ can be calculated by using a symbolic traversal algorithms similar to the ones described in chapter 4.

5.4 SYNCHRONOUS ABSTRACTION OF THE *TCSG*

This section explains how the *TCSG* is pruned in such a way that the input patterns that produce neither non-confluence nor oscillation are considered as valid candidates for test sequences. Roughly speaking, the *TCSG* will be reduced to a set of stable states and edges between stable states. For an edge (s, s') to exist, s must be stable and s' must be stable and the only state reached at the end of the test cycle. The finally obtained state graph will only contain the confluent and stable behavior of the original *TCSG*, hence the acronym *CSSG*, standing for *Confluent and Stable State Graph*.

We will use figure 5.7 as an example. Let us assume that $s1$ and $s2$ are initial states. A vector producing non-confluence is $A{+}C{-}$ applied to state $s1$, since either $s3$ or $s4$ can be nondeterministically reached. If vector $D+$ is applied to state $s4$ the circuit oscillates. The only valid vectors are $A{+}B{+}$ and $A{-}B{-}$ applied respectively to $s2$ and $s4$. This fact is manifested in figure 5.7(b). Note that the initial state $s1$ appears in the *CSSG*, even though no valid input pattern can be applied to it. Nevertheless, still some fault could be detected when forcing $s1$ as reset state.

5.4.1 Estimation of the test cycle

Unbounded gate delays and "long enough" test cycles are unrealistic assumptions for testing. Instead, bounded gate delays and short test cycles must be assumed. Moreover, the analysis of oscillation conditions is a difficult problem still under investigation.

Let σ be the longest sequence of transitions from a stable state s to the final stable state or states when certain input pattern is applied to s. If α is the longest gate delay, then $\tau = \alpha \cdot |\sigma|$ is an upper bound of the test cycle. On the contrary, if a test cycle of length t is desired, then $k = \lceil t/\alpha \rceil$ can be an estimation of the maximum number of allowed transitions before the circuit finally stabilizes. This is just an approximation we will use henceforth, but once the layout is provided, a more accurate cycle time can be calculated.

5.4.2 Practical computation of the *CSSG*

In order to calculate the *TCSG* synchronous abstraction, we first will define the pairs of states (s, s') such that s' is reached from s at the end of the test cycle. Each pair has an associated input pattern, given by the different values of inputs in s and s'. For the sake of clarity, subsequent pairs (s, s') will be assumed such that s is stable and s' is reached by propagating a single input pattern applied to s. We call the set of all these pairs of states the *test cycle relation*. For practical reasons we will assume that the circuit must settle in at most k transitions. The *k-step test cycle relation* (TCR^k) represents the pairs (s, s') distant at most k transitions. Formally, given a $TCSG$ $\langle \mathcal{S}, \mathcal{E}, \mathcal{P}, \mathcal{G}, S^0, \lambda_P, \lambda_G \rangle$, TCR^k is defined as:

$$TCR^k = \left\{ (s, s') \in \mathcal{S} \times \mathcal{S} \mid \exists s_1, \ldots, s_k \text{ such that } sR_I s_1 \ \wedge \ (\bigwedge_{i=2}^{k} s_{i-1} R_\delta s_i) \ \wedge \ s_k = s' \right\} .$$

The next step consists of removing invalid pairs of states. Vectors causing non-confluence are detected if pairs (s, s') and (s, s'') such that both s' and s'' have the same input values exist. Patterns producing oscillation or unacceptable long test cycle are found if s' is unstable. The *k-Confluent Stable State Graph*, denoted as $CSSG^k$, is formed by those pairs in TCR^k that neither present non-confluence nor cause the circuit to be unstable after k transitions. Formally, it can be defined as:

$$CSSG^k = \left\{ (s, s') \in TCR^k \mid s' \text{ is stable} \wedge \nexists (s, s'') \in TCR^k \text{ such that } [s' \neq s'' \ \wedge \ \lambda_I(s') = \lambda_I(s'')] \right\} .$$

Informally the $CSSG^k$ contains the following information. Each one of its nodes represents a stable state. An arc between two nodes s and s' exists if s' is stable and the only state reachable from s in at most k transitions by applying some input pattern.

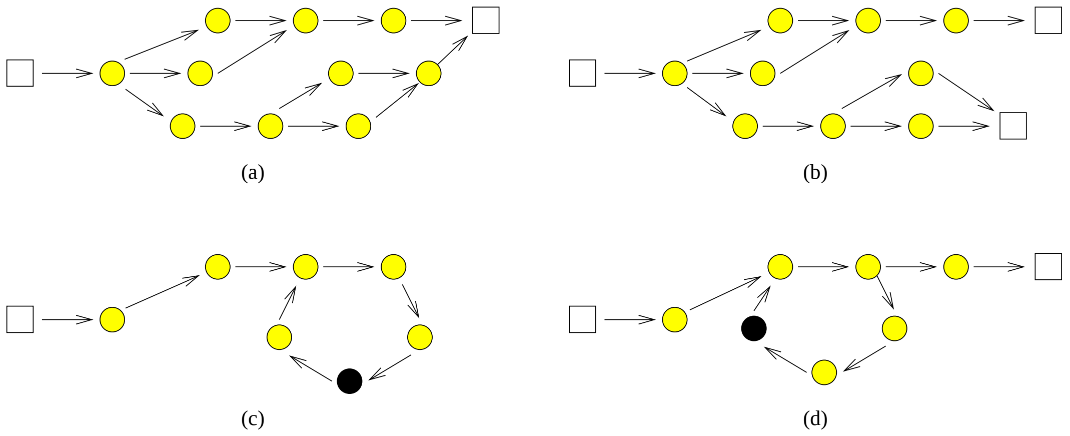


Figure 5.8 Examples of (a) confluence, (b) non-confluence, (c) non-transient cycle and (d) transient cycle.

Figure 5.8 illustrates the practical computation of the $CSSG$. Boxes represent stable circuit states, whereas circles represent transient states. Let us assume that the test cycle estimation is $k = 11$ transitions. Black circles symbolize the states reachable after exactly k transitions. In figures 5.8(a) and 5.8(b), the circuit always stabilizes in less than k transitions. In figure 5.8(a) the circuit always settles to a single state, while on the contrary, in figure 5.8(b) the circuit can nondeterministically stabilize in different states. Figure 5.8(c) shows a *non-transient cycle*, i.e. the circuit never reaches a stable state. This oscillation is easily noticed because after k transitions the state of the circuit is unstable. Finally, figure 5.8(d) presents a *transient cycle*. This situation is detected because after k transitions the circuit can be in either a stable or unstable state. The examples of non-confluence and oscillation given in section 5.3 correspond respectively to figures 5.8(b) and 5.8(c).

5.5 AUTOMATIC TEST PATTERN GENERATION

Many techniques have been proposed for *Automatic Test Pattern Generation* (ATPG) for sequential synchronous circuits. As we have been explaining, however, non-confluence of settling state and oscillation make that those techniques cannot be directly applied to asynchronous circuits. Our approach resembles the *Three-phase ATPG* [CHS91] proposed for synchronous circuits. We also propose a method with three phases: *fault activation*, *state justification* and *state differentiation*, described in sections 5.5.1 to 5.5.3. The way these three phases are implemented, though, will be different because of the asynchronous nature of circuits. Section 5.6 introduces Random TPG and fault simulation as techniques to increase the speed of the whole approach.

5.5.1 Fault activation

The first step to generate a test is to find a set of states that activate or excite the fault. It is easy to see that the fault signal x stuck-at-c is excited in some stable state if $x \neq c$. Since the set of stable reachable states has been already obtained during *CSSG* computation, finding the stable states exciting a fault is straightforward.

In most examples, there is always some stable state that excites a fault. However, it can occur that some signal always equals either 0 or 1 when the circuit is stable and only takes the opposite value in some unstable states. This situation arises when a signal switches an even number of times between stable states. Finding a test for such faults is left directly to the last phase, explained in section 5.5.3.

5.5.2 State justification

Justifying a state means to provide a sequence of input vectors that drive the circuit from the initial or reset state to that particular state. In our case, a sequence of test vectors that put the circuit in some of the excitation states must be given.

By using the reachability information it is easy to give a justification sequence. This sequence will put the correct circuit in a state that excites a given fault. However, the test vectors applied on the faulty circuit may result in a sequence of states that differs from that obtained in the correct one. In addition it has to be taken into account that some available sequences for the correct circuit can cause a faulty one to diverge or oscillate.

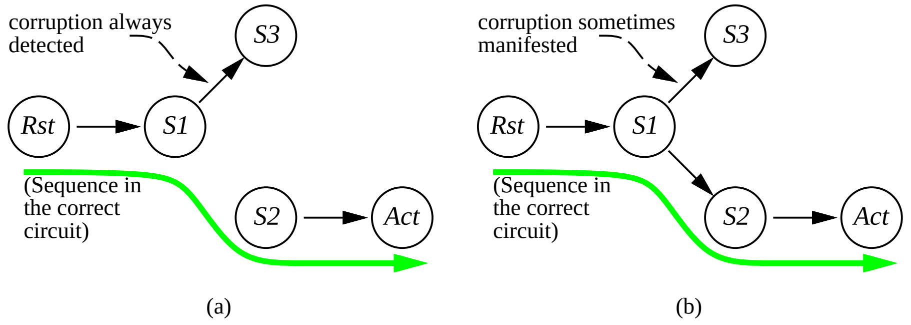

corruption always detected
S3
Rst
S1
(Sequence in the correct circuit)
S2
Act
(a)
corruption sometimes manifested
S3
Rst
S1
(Sequence in the correct circuit)
S2
Act
(b)

Figure 5.9 Corruption detected (a) always and (b) sometimes.

As noted in [CHS91], there could be *corruption*, so the fault would manifest before. In a synchronous circuit the sequence that produces the corruption can always be taken as a new shorter excitation sequence. However, in an asynchronous circuit corruption has to be noticed in all terminal stable states. If this symptom does not appear in some stable states, the entire sequence has to be applied. The consequence when testing the real circuit will be that sometimes the fault will be detected before others.

Figure 5.9 illustrates this by means of an example. According to the reachability analysis done in the correct circuit, the sequence of states $Rst \rightarrow S1 \rightarrow S2 \rightarrow Act$ is exercised. *Act* indicates the proposed activation state actually being justified. When the same inputs are applied to a faulty circuit the following situations might be observed. The stable state $S1$ in figure 5.9(a) has $S3$ as its successor, instead of $S2$ as in the correct circuit. Because of the different behavior of the correct and the faulty circuit, the fault can be detected before expected. The case in figure 5.9(b) is different. Now depending on the delays of the gates, two states, $S2$ and $S3$, are reachable from $S1$. Since the fault can not always be detected, we have to apply the full sequence of input vectors. During the real test operation however, the fault may be detected before.

5.5.3 State differentiation

Once a fault has been excited, it still has to be made observable. The most favorable case occurs when the fault is propagated to some primary output. In general, the fault will propagate to some memory element. By applying successive input vectors we have to make the difference noticeable at a primary output. The *CSSG* gives all the feasible input vectors that can be used in each state. All of them are simulated by using similar techniques to the ones described in section 5.7, and the sequence resulting in a shorter test length is chosen.

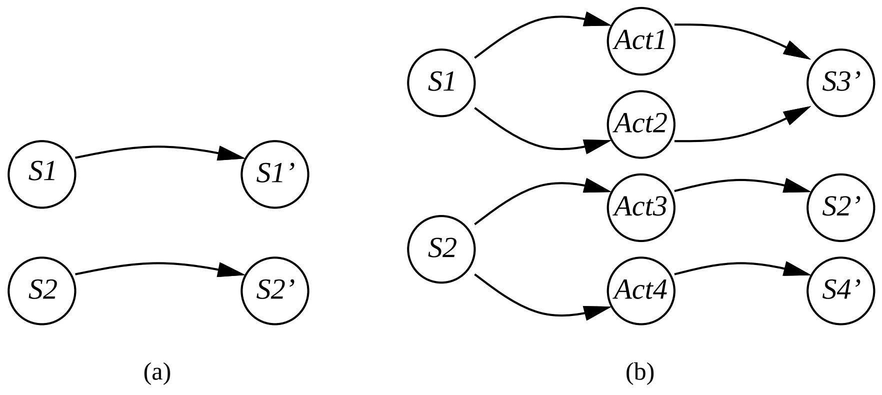


Figure 5.10 (a) Correct circuit (b) Faulty circuit.

As an example we can use figure 5.10. The different Act_i are the fault activating states, while $S1$ and $S2$ are their stable predecessors. In the correct circuit, $S1'$ and $S2'$ are reached, respectively, from $S1$ and $S2$. In the faulty circuit, $S3'$ is always reached from $S1$, therefore there is an appropriate excitation vector. However, depending on the gate delays, from $S2$ the fault can either be detected ($S2'$) or not ($S4'$). In the latter case, the test would not be conclusive.

5.6 IMPROVING ATPG PERFORMANCE

Using an ATPG algorithm like the one described in the previous section is sufficient to find a test vector for any testable fault. However, such a solution may be time consuming. For synchronous circuits it is traditionally considered that ATPG algorithms are more costly than simulation algorithms. This assumption is true if neither non-confluence nor oscillation occur, which is the usual case in synchronous circuits. For asynchronous circuits, in which such problems are almost unavoidable, simulation considering binary values can be as costly as test generation. Fortunately, if circuits are considered to work in ternary logic rather than boolean, i.e. signals can have the values 0, 1 or Φ, efficient although conservative simulation algorithms can be used. Taking that into account, techniques that improve the speed of synchronous ATPG algorithms can be adapted to asynchronous ones.

5.6.1 Ternary simulation

The state-explosion problem, that complicates asynchronous circuit synthesis, verification and test generation, also makes simulation difficult. In addition, non-confluence and oscillation could make unfeasible techniques like *event driven simulation* [Ulr65, Ulr69], that properly work with synchronous circuits. Fortunately, if circuit signals are considered to have ternary values instead of boolean, a very efficient method called *ternary simulation* [Eic65, BS95], can be used. With this technique simulation becomes polynomial in the number of circuit signals, but at the cost of being conservative.

```
Algorithm_A(($p,g$), $p'$) {
   for $i = 1,\dots,|\mathcal{P}|$ {
      $p_i$ := $lub(p_i, p'_i)$;
   }
   do {
      $g'$ := $g$;
      for $i = 1,\dots,|\mathcal{G}|$ {
         $g_i$ := $lub(g_i, f_{g_i}(p,g))$;
      }
   } while $g \neq g'$;
   return $(p,g)$;
}
```

```
Algorithm_B(($p,g$), $p'$)
   for $i = 1,\dots,|\mathcal{P}|$ {
      $p_i$ := $p'_i$;
   }
   do {
      $g'$ := $g$;
      for $i = 1,\dots,|\mathcal{G}|$ {
         $g_i$ := $f_{g_i}(p,g)$;
      }
   } while $g \neq g'$;
   return $(p,g)$;
}
```

Figure 5.11 Algorithms A and B making up ternary simulation.

In ternary simulation a signal can have any of the three following values: 0, 1 or Φ. The symbols 0 and 1 have their usual boolean meaning, whereas the symbol Φ stands for an uncertain value which is neither 0 nor 1 (see section 2.5).

Ternary simulation consists of two algorithms namely A and B (see figure 5.11). Algorithm A sets each signal to the least upper bound (*lub*) of its current value and its evaluation. The result is that unstable signals are set to Φ. By repeating this process, uncertainties are propagated through all the circuit. Algorithm B sets each signal to its evaluation. Consequently, some signals are set to a known value (either 0 or 1). Let us assume the circuit is in state s and we apply input vector a. After algorithms A and B we reach the final state s'. We can conclude that if all the signals in s' have a definite value (0 or 1), s' is the only successor of s when a is applied. On the contrary, if there are several final stable states or the circuit oscillates, some signal in s' will have an unknown value (Φ).

Ternary simulation is conservative because of the loss of information introduced by the Φ uncertain value [Bre72]. The best explanation to this problem is a simple example like the one in figure 5.12. Clearly, the function driving signal b is $a \cdot \bar{a}$, which is identically zero for any value of a. However, if a is set to Φ, b is consequently set to $\Phi \cdot \bar{\Phi} = \Phi$.

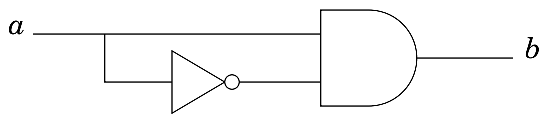


Figure 5.12 Circuit illustrating why ternary simulation is conservative.

It has been proved that ternary simulation is polynomial in the number of circuit gates [BS95]. This is due to the fact that in the worst case $2n$ states are produced, n being the number of gates. In each state at most n function evaluations are required. Therefore, detection of critical races and/or oscillation can be detected in $O(n^2)$ for each pair of stable state and input pattern.

5.6.2 Fault simulation

Fault simulation is commonly used to tell whether a test for a given fault also detects other faults. When a test is found to detect a fault, the same input patterns are simulated on the remaining faulty circuits. This technique will be efficient only if fast simulation algorithms are provided. Symbolic algorithms are good at managing multiple states of a same circuit, but the problem when simulating a fault is just the opposite: dealing with one state for each different faulty circuit. This problem can be solved by *parallel fault simulation* [Ses65].

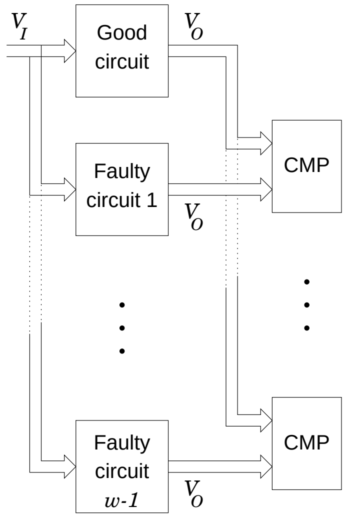
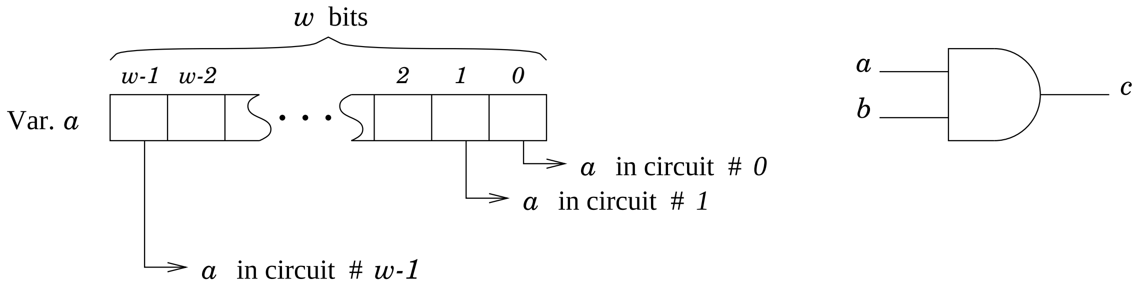


Figure 5.13 Parallel fault simulation handles several circuits simultaneously.

Parallel fault simulation exercises the good and a number of faulty circuits concurrently, as figure 5.13 illustrates. Each variable (signal, wire) of a circuit is packed in a same word together with the same variable corresponding to other circuits, and every bit of that variable corresponds to a different circuit (see figure 5.14). The bit 0 usually represents the variable in the good circuit, while the other bits keep the variable for the different faulty circuits. Since computers can perform operations like AND, OR, NOT, XOR, etc, with words of w bits, executing one of those instructions will produce a change in all w circuits. Stuck-at faults are simulated by inserting the stuck value in the corresponding bit of the stuck variable. Whether a fault is detected or not can be noticed by looking which bits of output signals differ from the bit 0.

As an example, let us take the AND gate in figure 5.14, with inputs a and b, and output c. The previous values of signals a, b and c can be seen in the table on the same figure ($w = 8$ bits is



	previous values	c := a AND b	circuit #*1* c stuck-at-0	circuit #*2* c stuck-at-1
a	01101101	01101101	01101101	01101101
b	10101010	10101010	10101010	10101010
c	11110010	00101000	00101000	00101100

Figure 5.14 Example of one step of parallel fault simulation.

assumed). In one single operation the value of c in all 8 circuits is computed. If we assume that circuit #1 has the fault c stuck-at-0, and circuit #2 c stuck-at-1, two new operations are required to set the bits 1 and 2 of variable c to 0 and 1 respectively. If variables represent signals, only faults at the gate outputs will be taken into account (as in this example). The input stuck-at model can be considered by assigning variables to the output of the gates and also to each end of a fork.

We use a combination of ternary simulation and parallel fault simulation in order to simulate the same input pattern in w different circuits. In this case, at least two bits are needed to encode the three different ternary values. A possible encoding appears in the next table

value	encoding
0	00
1	01
Φ	10

We must say that the inherent conservativeness of ternary simulation does not affect the fault coverage of our approach. The only loss will be in terms of time. The objective of fault simulation is just to find out when the same test detects some other faults. If ternary simulation says that a test for a given fault is unable to cover other faults, when in fact it could, tests for those faults will be found by the ATPG algorithm.

5.6.3 Random TPG

Random Test Pattern Generation (Random TPG) has turned out as a very efficient method in finding a test for an important number of faults at a very low CPU cost. The number of faults covered by this technique highly depends on the circuit, but coverage ratios between 40% and 80% are commonly achieved. By using ternary simulation with Random TPG the speed of the overall approach can be improved in similar percentages.

A random number of random test vectors is applied and parallel ternary simulation is used to decide which faults are detected by a given test vector. A uniform signal change probability is assumed, with the only exception of the RESET signal, that may become active with a probability smaller than one half. New random test patterns are applied as long as new faults are being covered.

In order to choose a new random test vector, we follow the approach by Breuer [Bre71]. Given a new test vector, in [Bre71] it is defined a cost function that depends on the number of new faults detected, propagated and "lost", i.e. faults that were propagated but applying the new vector eliminates those effects. This function is evaluated for a certain number of candidate vectors, and the one resulting in a higher value is chosen. This represents a local optimization technique.

5.7 RESULTS

We show the effectiveness of our ATPG methodology over a set of benchmarks. Table 5.1 presents the results obtained for speed-independent and table 5.2 for hazard-free circuits with bounded delays. Both sets of benchmarks have been automatically synthesized from the same specifications, the former by Petrify [CKK$^+$96] and the latter by SIS [SSL$^+$92].

Table 5.1 Experimental results (speed-independent).

	output-stuck-at		input-stuck-at					
example	total	covered	total	covered	rnd	3-phase	sim	CPU
alloc-outbound	32	32	66	66	51	12	3	52
atod	26	26	40	40	36	1	3	5
chu150	26	26	52	50	50	0	0	10
converta	22	22	44	44	20	23	1	8
dff	20	20	44	40	7	33	0	18
ebergen	32	32	70	70	32	38	0	43
hazard	20	20	44	44	22	22	0	7
master-read	62	62	130	130	55	75	0	5049
mmu	60	60	136	136	44	92	0	21067
mp-forward-pkt	28	28	58	58	57	1	0	6
mr1	60	60	140	139	5	134	0	27231
nak-pa	40	40	80	80	68	5	7	43
nowick	28	28	54	54	54	0	0	4
ram-read-sbuf	42	42	82	82	54	26	2	465
rcv-setup	20	20	36	36	31	5	0	4
rpdft	32	32	62	62	60	1	1	14
sbuf-ram-write	50	50	102	102	40	60	2	1760
sbuf-send-ctl	40	40	86	86	45	41	0	254
sbuf-send-pkt2	40	40	116	116	31	85	0	7137
seq4	40	40	86	84	28	52	4	256
trimos-send	58	58	132	132	6	126	0	16030
vbe10b	50	50	114	110	22	88	0	5534
vbe5b	22	22	42	41	33	8	0	9
vbe6a	38	38	80	78	17	61	0	562
Total FC	100.00%		99.16%					

The results in the tables are structured as follows. The second and third columns respectively present the *total* and *covered* number of faults under the single output stuck-at fault model. The fourth and fifth columns show analogous results for the single input stuck-at fault model. The next three columns, namely "rnd", "3-phase" and "sim", detail the number of faults covered by each step of our approach. The last column reports the CPU time, in seconds, needed to find the whole set of test vectors. The benchmarks have been run on a Sun 4 workstation with a SPARCStation-20 processor and 64 megabytes of RAM.

The input stuck-at fault model includes all output stuck-at faults. The results on output stuck-at faults are shown to illustrate that the well known theoretical result of speed-independent circuits being 100% output stuck-at fault testable in operation mode [BM92] still holds when our methodology is used.

Table 5.2 Experimental results (hazard free with bounded delays).

	output-stuck-at		input-stuck-at					
example	total	covered	total	covered	rnd	3-phase	sim	CPU
atod	44	44	66	66	52	11	3	487
chu150	26	26	48	46	44	2	0	15
converta	42	42	92	92	8	84	0	5008
ebergen	20	20	42	42	35	7	0	5
half	30	30	42	42	11	30	1	76
hazard	30	30	46	44	39	4	1	20
nowick	28	28	54	54	54	0	0	4
rpdft	36	36	48	48	48	0	0	8
trimos-send	72	24	126	29	12	17	0	254851
vbe10b	60	16	136	26	21	5	0	26774
vbe5b	32	32	52	52	42	10	0	19
vbe5c	24	24	36	36	34	2	0	4
vbe6a	62	18	126	29	23	6	0	29647
Total FC	73.12%		66.30%					

Conversely, this is not true for the set of circuits generated by SIS. Most circuits present similar results to those of speed-independent circuits, but three benchmarks, *trimos-send, vbe10b* and *vbe6a*, presented a very poor fault coverage. This is due to the logic redundancies added by the synthesis tools in order to avoid spurious pulses in this type of circuits. Note that these examples also take a very long time to finish. When a test for an undetectable fault is searched, all possible input patterns are tried, thus time is wasted with no positive results. Finding out *a priori* undetectable faults may result in significant performance increase. In those cases with very low fault coverage, testability can be assisted by *partial scan-path* [KB95] or *variable phase splitting* [LKL94].

The number of faults detected by random TPG depends highly on the example topology, but an average of 45% is achieved. This fact represents an important speed-up of our methodology. If a low coverage is achieved in the random step, much work will be left to the 3-phase step. 3-phase ATPG (fault activation, state justification and state differentiation) is the most complex step and the one dominating CPU time. Note that the highest test generation times correspond to those benchmarks where the random TPG step has covered a low number of faults (see e.g. *converta* and *trimos-send* in table 5.1). In some cases the same vector is reported to cover different faults. Due to the conservativeness of ternary simulation, it sometimes fails to detect equivalent tests. This is the reason for the low number of faults covered by fault simulation. Despite the the low number of faults covered by fault simulation, this last step is still performed because its execution time is negligible when compared to the 3-phase ATPG algorithm.

As a general consideration, such results can be significantly improved by speeding up the 3-phase step. Three possibilities we have in mind are: studying better variable ordering strategies in the use of BDDs, using hierarchical techniques similar to those utilized in some formal verification approaches [RCP95a] and characterizing undetectable faults to avoid wasting time in covering them.

5.7.1 Discussion

Banerjee et al. [BCR96] also propose synchronous testing of asynchronous circuits. They model the asynchronous circuit as a synchronous one by cutting feedback loops by virtual synchronous flip-flops. Hence, ATPG can be done by using efficient state-of-the-art synchronous techniques. Test vectors are validated afterwards on the asynchronous circuit by using zero-delay and unit-delay simulation [Ban97]. Clearly, that validation can detect oscillation, but it is unable to identify non-confluence. This causes their approach to be optimistic.

Our approach assumes the pessimistic unbounded gate delay model, which assures that test vectors generated with our methodology are independent from the technology and the gate delays. Another difference between the method in [BCR96] and ours is that we analyze the asynchronous circuit to find out those vectors that can be used in ATPG, so no further validation is needed. This analysis is done on the asynchronous circuit, rather than on a synchronous simplification. Consequently, our approach is computationally more expensive, but it can cope naturally with oscillation and non-confluence. Possibly, a hybrid method could take the best of both approaches.

5.8 CONCLUSIONS

Testing of asynchronous circuits is a problem still far from being solved satisfactorily. This chapter has presented a method for ATPG based on well-known techniques for synchronous circuits.

The results shown in this chapter indicate that asynchronous control circuits are highly testable without applying partial scan techniques. Automatic techniques to select those signals in which the insertion of scan paths can contribute to improve testability is also one of the goals to be pursued in the future.

The main contribution of this work is the synchronous abstraction performed over an asynchronous system such that real-life synchronous testers can be used to exercise the input signals.

This is only a preliminary work that will be further developed towards covering a wider spectrum of fault models (e.g. delay faults) with more efficient approaches. In the near future we want to explore the possibility of using hierarchy to tackle the testing of complex asynchronous systems. The synchronous abstraction of the circuit's behavior allows partitioning of large circuits into several interacting asynchronous circuits. We believe this feature will help to generate test patterns with techniques based on the composition of finite state machines.

6 CONCLUSIONS AND FUTURE WORK

NAPOLEON: What shall we do with this soldier, Guiseppe? Everything he says is wrong.
GUISEPPE: Make him a general, Excellency, and then everything he says will be right.
– G. B. Shaw, "The Man of Destiny".

Obró mucho el que nada dejó para mañana.
– Baltasar Gracián.

Asynchronous circuits offer potential advantages that make them an interesting design option for high-speed low-power digital systems. Unfortunately, they are more difficult to design than their synchronous counterparts. Hundreds of CAD packages are available to ease synchronous design, but it has not been until very recently that the first experimental tools for asynchronous design have begun to appear.

This work is a small step towards the ease of asynchronous design. Two crucial points have been addressed in order to guarantee that neither incorrect nor faulty asynchronous circuits come to the market: formal verification and testing. The promising results in both fields deserve new research attention for increasing the features of the presented approaches.

6.1 MAIN CONTRIBUTIONS

The main contributions that have been presented are the following:

- Efficient manipulation of safe Petri nets and speed-independent circuits by using BDDs.
- Verification of asynchronous systems:
  - Formal verification of Petri net properties.
  - Formal verification of speed-independence by composing a circuit with its Petri net specification (or environment).
  - Formal verification of properties in closed environment-circuit systems.
  - Simplification of verification complexity by abstracting most combinational internal signals.
- Testing of asynchronous circuits:
  - Characterization of asynchronous circuit behavior when synchronous input vectors are applied.

   – Automatic test pattern generation for single stuck-at faults by using synchronous test vectors.

- The above contributions have been implemented in form of automatic asynchronous circuit verification and test generation tools, respectively *versify* and *testify*. Both tools, as well as several examples, are available through WWW from the URLs:

  `http://www.ac.upc.es/recerca/VLSI/versify`
  `http://www.ac.upc.es/recerca/VLSI/testify`

6.2 OPEN PROBLEMS AND FUTURE WORK

The experimental results achieved by applying symbolic techniques to automatic verification and test pattern generation are encouraging. However, more effort must be devised in order to extend the presented methodologies to other classes of asynchronous circuits and different fault models.

6.2.1 Verification

The presented method is addressed to verify specifications and circuits designed assuming unbounded delays. By using more realistic delay models the accuracy of verification may be significantly increased. In order to verify designs that suppose timing restrictions, *bounded delays* should be used.

Possibly, in a recent future designs will be implemented in a mixed asynchronous-synchronous fashion. The circuits will be locally synchronous, but will use asynchronous communication to interact. Verification of these hybrid systems will also be considered.

6.2.2 Testing

Unlike in verification, we have seen in chapter 5 that the unbounded gate delay assumption is not a problem when testing. Nevertheless, the contrived topology of asynchronous circuits implies important modifications to the traditional ATPG algorithms. Hybrid structural-symbolic methodologies could probably be more efficient than our symbolic ATPG algorithm. Also, characterizing untestable faults will significantly speed up the whole ATPG process.

Other fault models should also be taken into account in the future, e.g. bridging faults or path delay faults, etc. Finally, similar methodologies could be used in order to take advantage of the emerging IDD testing.

REFERENCES

[Ban97] Savita Banerjee. Personal communication, March 1997.

[BB95] Peter A. Beerel and Jerry R. Burch. Personal communication, April 1995.

[BBM94] Peter A. Beerel, Jerry R. Burch, and Teresa H.-Y. Meng. Sufficient conditions for correct gate-level speed-independent circuits. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 33–43. IEEE Computer Society Press, November 1994.

[BBR90] K. S. Brace, R. E. Bryant, and R. L. Rudell. Efficient implementation of a BDD package. In *Proceedings of the 27th Design Automation Conference*, pages 40–45, 1990.

[BCL$^+$94] Jerry R. Burch, Edmund M. Clarke, D. E. Long, Kenneth L. McMillan, and David L. Dill. Symbolic model checking for sequential circuit verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(4):401–424, 1994.

[BCR96] Savita Banerjee, Srimat T. Chakradhar, and Rabindra K. Roy. Synchronous test generation model for asynchronous circuits. In *Proc. of the International Conference on VLSI Design*, Bangalore, January 1996.

[BM92] Peter A. Beerel and Teresa H.-Y. Meng. Semi-modularity and testability of speed-independent circuits. *Integration, the VLSI journal*, 13(3):301–322, September 1992.

[Boh96] Mark T. Bohr. Interconnect scaling – the real limiter to high performance ULSI. *Solid State Technology*, pages 105–111, September 1996.

[Boo54] G. Boole. *An Investigation of the Laws of Thought.* Walton, London, 1854. (Reprinted by Dover Books, New York, 1954).

[Bre71] Melvin A. Breuer. A random and an algorithmic technique for fault detection test generation for sequential circuits. *IEEE Transactions on Computers*, C-20(11):1364–1370, November 1971.

[Bre72] Melvin A. Breuer. A note on three valued logic simulation. *IEEE Transactions on Computers*, C(21):399–402, April 1972.

[Bre74] Melvin A. Breuer. The effects of races, delays, and delay faults on test generation. *IEEE Transactions on Computers*, C-23(10), October 1974.

[Bro90] F. M. Brown. *Boolean Reasoning: The Logic of Boolean Equations.* Kluwer Academic Publishers, 1990.

[Bry86] R.E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[BS95] Janusz A. Brzozowski and Carl-Johan H. Seger. *Asynchronous Circuits.* Monographs in Computer Science. Springer-Verlag, 1995.

[Bur90] Jerry R. Burch. Verifying liveness properties by verifying safety properties. In Robert P. Kurshan and Edmund M. Clarke, editors, *Proc. International Workshop on Computer Aided Verification*, volume 531 of *Lecture Notes in Computer Science*, pages 224–232. Springer-Verlag, 1990.

[Cad89] Cadence Design Systems. *Verilog XL Reference Manual*, September 1989.

[CBM89] O. Coudert, C. Berthet, and J. C. Madre. Verification of sequential machines using boolean functional vectors. In *Proc. IFIP Int. Workshop on Applied Formal Methods for Correct VLSI Design*, pages 111–128, Leuven, Belgium, November 1989.

[CE81] E. M. Clarke and E. A. Emerson. Synthesis of synchronization skeletons for branching time temporal logic. In Dexter Kozen, editor, *Logic of Programs: Workshop*, volume 131 of *Lecture Notes in Computer Science*, 1981.

[CHS91] Hyunwoo Cho, Gary D. Hachtel, and Fabio Somenzi. Fast sequential ATPG based on implicit state enumeration. In *Proc. International Test Conference*, pages 67 – 74, Nashville, TN, October 1991.

[Chu86] T.-A. Chu. Synthesis of self-timed control circuits from graphs: An example. In *Proc. of the IEEE International Conference on Computer Design*, pages 565–571, October 1986.

[Chu87] T.-A. Chu. *Synthesis of Self-timed VLSI Circuits from Graph-theoretic Specifications.* PhD thesis, MIT, June 1987.

[CKK+96] Jordi Cortadella, Michael Kishinevsky, Alex Kondratyev, Luciano Lavagno, and Alexandre Yakovlev. Petrify: a tool for manipulating concurrent specifications and synthesis of asynchronous controllers. In *XI Conference on Design of Integrated Circuits and Systems*, Barcelona, November 1996.

[DC86] David L. Dill and Edmund M. Clarke. Automatic verification of asynchronous circuits using temporal logic. *IEE Proceedings, Part E, Computers and Digital Techniques*, 133:272–282, September 1986.

[DE95] J. Desel and J. Esparza. *Free Choice Petri Nets.* Cambridge University Press, Cambridge, Great Britain, 1995.

[DGY95] Ilana David, Ran Ginosar, and Michael Yoeli. Self-timed is self-diagnostic. *Journal of Electronic Testing:Theory and Applications*, (6):219–228, January 1995.

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

[ea95] W.J. Bowhill et al. A 300 MHz 64-b quad-issue CMOS RISC microprocessor. In *International Solid State Circuits Conference*, pages 182–183, February 1995.

[Ebe89] J. C. Ebergen. *Translating Programs into Delay-Insensitive Circuits*, volume 56 of *CWI Tract.* Centre for Mathematics and Computer Science, 1989.

[Eic65] E. B. Eichelberger. Hazard detection in combinational and sequential switching circuits. *IBM Journal of Research and Development*, 9:90–99, March 1965.

[FDG+93] S. B. Furber, P. Day, J. D. Garside, N. C. Paver, and J. V. Woods. A micropipelined ARM. In T. Yanagawa and P. A. Ivey, editors, *Proceedings of VLSI 93*, pages 5.4.1–5.4.10, September 1993.

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

[Gor85] Mike Gordon. Why higher-order logic is a good formalism for specifying and verifying hardware. In *Formal Aspects of VLSI Design*, pages 153–177. North Holland, 1985.

[Haz92] Pieter J. Hazewindus. *Testing Delay-Insensitive Circuits.* PhD thesis, California Institute of Technology, 1992.

[HHY92] Kiyoharu Hamaguchi, Hiromi Hiraishi, and Shuzo Yajima. Design verification of asynchronous sequential circuits using symbolic model checking. In *International Symposium on Logic Synthesis and Microprocessor Architecture*, pages 84–90, July 1992.

[Hoa78] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985.

[Huf64] D. A. Huffman. The synthesis of sequential switching circuits. In E. F. Moore, editor, *Sequential Machines: Selected Papers.* Addison-Wesley, 1964.

[JQN$^+$91] B. Johnson, T. Quarles, A.R. Newton, D.O. Pederson, and A. Sangiovanni-Vincentelli. *SPICE3 Version 3e User's Manual.* Dept. of Electrical Engineering and Computer Science, Univ. of California, Berkeley, April 1991.

[KB95] Ajay Khoche and Erik Brunvand. Testing self-timed circuits using partial scan. In *Proc. of the 2nd Working Conference on Asynchronous Design Methodologies*, pages 160–169, London, May 1995.

[KCK$^+$95] Alex Kondratyev, Jordi Cortadella, Michael Kishinevsky, Enric Pastor, Oriol Roig, and Alexandre Yakovlev. Checking signal transition graph implementability by symbolic BDD traversal. In *Proc. European Design and Test Conference (EDAC-ETC-EuroASIC)*, pages 325–332, Paris, March 1995.

[KKTV94a] M. Kishinevsky, A. Kondratyev, A. Taubin, and V. Varshavsky. Analysis and identification of speed-independent circuits on an event model. *Formal Methods in System Design*, 4(1):33–75, January 1994.

[KKTV94b] M. Kishinevsky, A. Kondratyev, A. Taubin, and V. Varshavsky. *Concurrent Hardware. The Theory and Practice of Self-timed Design.* Series in Parallel Computing. John Wiley & Sons, 1994.

[KLSV91] K. Keutzer, Luciano Lavagno, and A. Sangiovanni-Vincentelli. Synthesis for testability techniques for asynchronous circuits. In *Proc. of the IEEE/ACM International Conference on Computer Aided Design*, pages 326–329. IEEE Computer Society Press, November 1991.

[KLSV95] K. Keutzer, Luciano Lavagno, and A. Sangiovanni-Vincentelli. Synthesis for testability techniques for asynchronous circuits. *IEEE Transactions on Computer-Aided Design*, 11(1):87–101, December 1995.

[Kur86] Robert P. Kurshan. Testing containment of ω-regular languages. Technical Report 1121-861010-33-TM, Bell Laboratories, 1986.

[Kur87] Robert P. Kurshan. Reducibility in analysis of coordination. In *LNCS*, volume 103, pages 19–39. Springer-Verlag, 1987.

[Lav92] Luciano Lavagno. *Synthesis and Testing of Bounded Wire Delay Asynchronous Circuits from Signal Transition Graphs.* PhD thesis, University of California at Berkeley, 1992.

[Lee59] C. Y. Lee. Representation of switching circuits by binary-decision programs. *Bell Technical Journal 38*, pages 985–999, 1959.

[LKL94] Luciano Lavagno, Michael Kishinevsky, and Antonio Lioy. Testing redundant asynchronous circuits. In *Proc. European Design Automation Conference (EURO-DAC)*. IEEE Computer Society Press, September 1994.

[LL92] H-T. Liaw and C-S. Lin. On the OBDD representation of generalized boolean functions. *IEEE Transactions on Computers*, 41(6):661–664, June 1992.

[Lon93] David E. Long. *A binary decision diagram (BDD) package*, June 1993. Manual page.

[LR87] Chin J. Lin and Sudhakar M. Reddy. On delay fault testing in logic circuits. *IEEE Transactions on Computer-Aided Design*, 6(5), September 1987.

[Mar85] Alain J. Martin. The design of a self-timed circuit for distributed mutual exclusion. In Henry Fuchs, editor, *Proceedings of the Chapel Hill Conference on VLSI*, pages 245–260. Computer Science Press, 1985.

[Mar86] Alain J. Martin. Self-timed FIFO: An exercise in compiling programs into VLSI circuits. In D. Borrione, editor, *From HDL Descriptions to Guaranteed Correct Circuit Designs*, pages 133–153. Elsevier Science Publishers, 1986.

[Mar90a] Alain J. Martin. The limitations to delay-insensitivity in asynchronous circuits. In William J. Dally, editor, *Sixth MIT Conference on Advanced Research in VLSI*, pages 263–278. MIT Press, 1990.

[Mar90b] Alain J. Martin. Programming in VLSI: From communicating processes to delay-insensitive circuits. In C. A. R. Hoare, editor, *Developments in Concurrency and Communication*, UT Year of Programming Series, pages 1–64. Addison-Wesley, 1990.

[MB59] D.E. Muller and W.C. Bartky. A Theory of Asynchronous Circuits. In *Annals of Computing Laboratory of Hardward University*, pages 204–243, 1959.

[McM92] Kenneth L. McMillan. Using unfoldings to avoid the state explosion problem in the verification of asynchronous circuits. In G. v. Bochman and D. K. Probst, editors, *Proc. International Workshop on Computer Aided Verification*, volume 663 of *Lecture Notes in Computer Science*, pages 164–177. Springer-Verlag, 1992.

[MFR85] Charles E. Molnar, Ting-Pien Fang, and Frederick U. Rosenberger. Synthesis of delay-insensitive modules. In Henry Fuchs, editor, *Proceedings of the Chapel Hill Conference on VLSI*, pages 67–86, 1985.

[MH91] Alain J. Martin and Pieter J. Hazewindus. Testing delay-insensitive circuits. In Carlo H. Séquin, editor, *Advanced Research in VLSI: Proceedings of the 1991 UC Santa Cruz Conference*, pages 118–132. MIT Press, 1991.

[Mil80] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag, 1980.

[Min96] Shin-ichi Minato. *Binary Decision Diagrams and Applications for VLSI CAD*. Kluwer Academic Publishers, 1996.

[Mul63] David E. Muller. Asynchronous logics and application to information processing. In *Symposium on the Application of Switching Theory to Space Technology*, pages 289–297. Stanford University Press, 1963.

[Mur89] Tadao Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–574, April 1989.

[New88] New York: The Institute of Electrical and Electronic Engineers, Inc. *IEEE Standard VHDL Language Reference Manual*, March 1988.

[Now93] Steven M. Nowick. *Automatic Synthesis of Burst-Mode Asynchronous Controllers.* PhD thesis, Stanford University, Department of Computer Science, 1993.

[NUK+94] Takashi Nanya, Yoichiro Ueno, Hiroto Kagotani, Masashi Kuwako, and Akihiro Takamura. TITAC: Design of a quasi-delay-insensitive microprocessor. *IEEE Design & Test of Computers*, 11(2):50–63, 1994.

[Pas96] Enric Pastor. *Structural Methods for the Synthesis of Asynchronous Circuits from Signal Transition Graphs.* PhD thesis, Universitat Politècnica de Catalunya, February 1996.

[PC93] Enric Pastor and Jordi Cortadella. Polynomial algorithms for the synthesis of hazard-free circuits from signal transition graphs. In *Proc. of the IEEE/ACM International Conference on Computer Aided Design*, pages 250–254, Santa Clara, USA, November 1993. IEEE Computer Society Press.

[Pet62] C. A. Petri. *Kommunikation mit Automaten.* PhD thesis, Bonn, Institut für Instrumentelle Mathematik, 1962. (technical report Schriften des IIM Nr. 3).

[Pet81] James L. Peterson. *Petri Net Theory and the Modeling of Systems.* Prentice-Hall, 1981.

[PRC97] Enric Pastor, Oriol Roig, and Jordi Cortadella. Symbolic Petri net analysis using boolean manipulation. Technical Report UPC–DAC–97–8, Departament Arquitectura de Computadors (UPC), 1997.

[PRCB94] Enric Pastor, Oriol Roig, Jordi Cortadella, and Rosa Badia. Petri net analysis using boolean manipulation. In *15th International Conference on Application and Theory of Petri Nets*, volume 815 of *Lecture Notes in Computer Science*, pages 416–435. Springer-Verlag, June 1994.

[QS81] J. P. Quielle and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *Proc. of the Fifth International Symposium in Programming*, 1981.

[RCP95a] Oriol Roig, Jordi Cortadella, and Enric Pastor. Hierarchical gate-level verification of speed-independent circuits. In *Proc. of the 2nd Working Conference on Asynchronous Design Methodologies*, pages 128–137, London, May 1995.

[RCP95b] Oriol Roig, Jordi Cortadella, and Enric Pastor. Verification of asynchronous circuits by BDD-based model checking of Petri nets. In *16th International Conference on Application and Theory of Petri Nets*, volume 935 of *Lecture Notes in Computer Science*, pages 374–391, Torino, June 1995. Springer-Verlag.

[RCPP97] Oriol Roig, Jordi Cortadella, Marco A. Peña, and Enric Pastor. Automatic generation of synchronous test patterns for asynchronous circuits. In *Proceedings of the 34th Design Automation Conference*, June 1997. (To appear).

[RK93] D.S. Rao and F.J. Kurdahi. On clustering for maximal regularity extraction. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 12(8):1198–1208, August 1993.

[RMCF88] Fred U. Rosenberger, Charles E. Molnar, Thomas J. Chaney, and Ting-Pien Fang. Q-modules: Internally clocked delay-insensitive modules. *IEEE Transactions on Computers*, C-37(9):1005–1018, September 1988.

[RPC94] Oriol Roig, Enric Pastor, and Jordi Cortadella. Verificación de circuitos independientes de la velocidad con modelos simbólicos de redes de Petri. In *IX Congreso de Diseño de Circuitos Integrados*, pages 307–312, Las Palmas de Gran Canaria, November 1994.

[RS93] Marly Roncken and Ronald Saeijs. Linear test times for delay-insensitive circuits: a compilation strategy. In S. Furber and M. Edwards, editors, *Asynchronous Design Methodologies*, volume A-28 of *IFIP Transactions*, pages 13–27. Elsevier Science Publishers, 1993.

[RY85] L. Ya. Rosenblum and A. V. Yakovlev. Signal graphs: From self-timed to timed ones. In *International Workshop on Timed Petri Nets*, pages 199–206, July 1985.

[Sei80a] C. L. Seitz. System timing. In *Introduction to VLSI Systems*, chapter 7. Mead & Conway, Addison-Wesley, 1980.

[Sei80b] Charles L. Seitz. Ideas about arbiters. *Lambda*, 1(1, First Quarter):10–14, 1980.

[Ses65] S. Seshu. On an improved diagnosis program. *IEEE Transactions on Electronic Computers*, EC-12(2):76–79, February 1965.

[Sha49] C. E. Shannon. The synthesis of two-terminal switching circuits. *Bell System Technical Journal*, 28(1):59–98, 1949.

[Sil85] Manuel Silva. *Las Redes de Petri: en la Automática y la Informática.* Editorial AC, Madrid, 1985.

[Smi85] G. L. Smith. A model for delay faults based on paths. In *Proc. International Test Conference*, pages 324–349, September 1985.

[SSL+92] E. M. Sentovich, K. J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. R. Stephan, R. K. Brayton, and A. Sangiovanni-Vincentelli. SIS: A system for sequential circuits synthesis. Technical Report M92/41, UCB/ERL, May 1992.

[Sut89] I. E. Sutherland. Micropipelines. *Communications of the ACM*, June 1989. Turing Award Lecture.

[TSL+90] H. Touati, H. Savoj, B. Lin, R. K. Brayton, and A. Sangiovanni-Vincentelli. Implicit enumeration of finite state machines using BDD's. In *Proc. of the IEEE/ACM International Conference on Computer Aided Design*, pages 130–133, November 1990.

[Ulr65] E. G. Ulrich. Time-sequenced logic simulation based on circuit delay and selective tracing of active network paths. In *Proceedings of the 20th ACM National Conference*, pages 437–448, 1965.

[Ulr69] E. G. Ulrich. Exclusive simulation of activity in digital networks. *Communications of the ACM*, 13:102–110, February 1969.

[Van90] P. Vanbekbergen. Optimized synthesis of asynchronous control circuits from graph-theoretic specification. In *Proc. of the IEEE/ACM International Conference on Computer Aided Design*, pages 184–187, November 1990.

[vdS83] J. L. A. van de Snepscheut. *Trace Theory and VLSI design.* PhD thesis, Department of Computer Science, Eindhoven University of Technology, October 1983.

[YHTM96] T. Yoneda, H. Hatori, A. Takahara, and S. Minato. BDDs vs. Zero-Suppressed BDDs: for CTL symbolic model checking of petri nets. In *Proceedings of Formal Methods in Computer-Aided Design*, volume 1166 of *Lecture Notes in Computer Science*, pages 435–449. Springer-Verlag, 1996.

[YLSV92] A. Yakovlev, L. Lavagno, and A. Sangiovanni-Vincentelli. A unified signal transition graph model for asynchronous control circuit synthesis. In *Proc. of the IEEE/ACM International Conference on Computer Aided Design*, pages 104–111. IEEE Computer Society Press, November 1992.

LIST OF FIGURES

Chapter 1

1.1 Typical circuit design pipeline. 2
1.2 Universe of good/bad circuits given a fail/pass test. 4
1.3 Difference between pure and inertial delay. 8
1.4 (a) Feedback, (b) gate and (c) wire delay models. 8

Chapter 2

2.1 BDD example. 20
2.2 The `ite` algorithm. 22

Chapter 3

3.1 (a) A sample Petri net. (b) The same net after firing transition t_1. 24
3.2 (a) A Petri net and (b) its reachable markings. 26
3.3 Isomorphic (a) Petri net and (b) STG. 29
3.4 (a) Algorithm for symbolic traversal and (b) graphical representation of Breadth First Search. 32
3.5 Transformations preserving liveness, safeness and boundedness. 33
3.6 (a) Two consecutive transitions. Traversal (b) without or (c) with chaining. 34
3.7 (a)Traversal without chaining and (b) chaining. 35
3.8 BDD AND-toggle algorithm. 37
3.9 Improved BFS algorithm. 37
3.10 Improvements obtained by (a) chaining and (b) toggling techniques on the dining philosophers example. 38
3.11 (a) Firing t produces an unsafe marking. (b) Algorithm for safeness checking. 39
3.12 Algorithm to compute the SCC and TSCC sets of $[m_0\rangle$. 41
3.13 Algorithms to verify (a) transition persistence and (b) signal persistence. 41
3.14 Transition and signal (non-)persistence. 42
3.15 Algorithm for consistence checking. 42
3.16 Petri net for a dining philosopher. 44
3.17 Slotted ring protocol for one node. 45
3.18 Symbolic traversal algorithm: (a) number of states calculated and (b) size of the variables after each computation of δ (for the 8-node slotted ring example). 45

Chapter 4

4.1 Verification scenario: a circuit composed with an environment. 49
4.2 Modeling non-isochronic forks at (a) primary inputs and (b) gate outputs. 51
4.3 (a) Combinational NOR gate. (b) C memory element and (c) its internal representation. 52

4.4 Environment-circuit system. 53
4.5 Transition $a+$ is synchronized with switching a from 0 to 1. 54
4.6 (a) Circuit with a speed-independent behavior, (b) The same circuit with a hazardous behavior, (c) Equivalent speed-independent complex-gate circuit. 55
4.7 State Graph of the circuit in figure 4.6(a) (without place variables, for clarity). 55
4.8 Switching signal d from 1 to 0 is not accepted by the environment. 56
4.9 Basic BFS algorithm. 58
4.10 Stable AND gate becoming excited by backward switching. 59
4.11 Diagnosis algorithm. 59
4.12 (a) Environment-circuit system and (b) its complex-gate equivalent system. 60
4.13 (a) State graph after functional correctness verification. (b) Projection onto the interface signals of the complex gate. (c) Part of the state graph after behavioral correctness verification. 62
4.14 (a) Flat circuit (8 gates). (b) Hierarchical complex-gate organization (3 complex gates). 63
4.15 OR and C gates collapsed into a complex gate. 64
4.16 (a) Gate with multiple fanout. (b) Complex gate considered for functional correctness (c) and for behavioral correctness. 64
4.17 Example of (a) flat and (b) collapsed circuit. 64
4.18 (a) Environment of circuit. Projections (b) adding nondeterminism (c) or not. 65
4.19 Hierarchical verification algorithm 65
4.20 Union of environments for different instances of the same gate. 69
4.21 Algorithm for checking deadlock freeness. 71
4.22 Algorithm for checking the home-state property. 71
4.23 Two observationally equivalent circuits. 72
4.24 Comparison with Burch et al. 74

Chapter 5

5.1 Two different *short* faults mapping to the same stuck-at 1 fault. 81
5.2 Fault location in the output (a) and input (b) stuck-at models. 81
5.3 Examples of equivalent faults in NOR (a) and AND-OR (b) gates. 82
5.4 Goal of testing: observing the discrepancies between good and faulty circuits when the same patterns are applied. 82
5.5 Circuits showing (a) non-confluence of settling state and (b) oscillation. 83
5.6 State graph of a circuit showing non-confluence. 83
5.7 (a) A *TCSG* and (b) its corresponding *CSSG*. 85
5.8 Examples of (a) confluence, (b) non-confluence, (c) non-transient cycle and (d) transient cycle. 87
5.9 Corruption detected (a) always and (b) sometimes. 88
5.10 (a) Correct circuit (b) Faulty circuit. 89
5.11 Algorithms A and B making up ternary simulation. 90
5.12 Circuit illustrating why ternary simulation is conservative. 91
5.13 Parallel fault simulation handles several circuits simultaneously. 91
5.14 Example of one step of parallel fault simulation. 92

LIST OF TABLES

Chapter 3

3.1 Petri net reductions and their inverse transformations. 34
3.2 Results for the *dining philosophers* example. 44
3.3 Results for the slotted ring example. 45
3.4 STG verification results. 46

Chapter 4

4.1 Experimental results with speed-independent circuits. 73
4.2 Experimental results with circuits not satisfying its specification. 74

Chapter 5

5.1 Experimental results (speed-independent). 93
5.2 Experimental results (hazard free with bounded delays). 94

INDEX

Symbols

- Δ_N, 31
- Φ, 19, 90
- δ_G, 52
- δ_G^{-1}, 58
- δ_I, 54
- δ_I^{-1}, 58
- δ_N, 27, 36, 54
- δ_N^{-1}, 58
- δ_O, 54
- δ_O^{-1}, 58
- δ_S, 30, 36, 54
- λ_G, 84
- λ_P, 84
- λ
  - labeling function for states, 30, 66
  - labeling function for transitions, 29
- π, 30

A

- algebra, *see* Boolean or ternary algebra
- AND-Toggle operation, 36
- ASM$_t$, 28
- asynchronous circuits, 7
  - pros and cons, 5
- asynchronous microprocessor, 9
- ATPG, 10, 87
  - acceleration, 80
- Automatic Test Pattern Generation, *see* ATPG
- autonomous circuit, 50, 66
- average-case performance, 6

B

- B, 16
- B^n, 16, 26
- backward traversal, 58
- BDD, 10, 19
  - complementing variables, 36
  - of an equation, 21
  - representing sets, 25
- behavioral correctness, 60
- BFS
  - algorithm, 32, 58
  - improvements, 34
- Binary Decision Diagram, *see* BDD
- binary relation, *see* relation
- Boole's expansion, 21, 67
- Boolean algebra, 17
  - of n-variable logic functions, 26
  - of classes, 18
  - of logic functions, 18
  - of sets of markings, 26
  - of sets of STG states, 29
- bounded delay, 7, 98
- Breadth First Search, *see* BFS

C

- Cartesian product, 15, 53
- chaining, 34
- characteristic function, 26, 57
  - of a deadlock, 39, 70
  - of a relation, 27
  - of a set, 26
  - of states violating CSC, 43
  - of unsafe marking, 39
- choice, 41
- circuit, 49
  - autonomous, *see* autonomous circuit
  - levels of hierarchy, 69
- circuit design task flow, 1
- clock skew, 5
- codomain of a function, 16
- cofactor, 17, 28
- collapsed circuit, 63, 66, *see also* complex-gate-equivalent circuit
  - verification of properties, 71
- collapsing, 61, 63
  - example, 64
  - gates with multiple fanout, 64
- Complete State Coding, *see* CSC
- complex gate, 61
  - reduction to, 63
- complex-gate-equivalent circuit, 63, 71, *see also* collapsed circuit
- composition of relations, 15
- confluence, 87
- Confluent Stable State Graph, *see* CSSG
- conform to, 3, 48, 49, 56, 57

conservative test, 4
consistence, 41
constrained image computation, 27, 28, 39
  of a Petri net, 31
controllability, 77
counterexample, 58
CSC, 42
CSSG, 80, 86
  example, 85
cube, 16

D

dead signal, 72
dead transition, 39
deadlock, 39
deadlock freeness, 70
delay, 7, 51
delay-insensitive circuit, 8, 51
design for testability, 78
detectability, 77
diagnosis, 58
domain of a function, 16
dual-rail encoding, 5

E

E_t, 28
E_{v*}, 30
E_{v+}, 30
E_{v-}, 30
element of a set, 14
enabled transition, 23
encoding function, 26
environment-circuit system, 10
  verification of properties, 70
environment-circuit sytem, 53
equivalence relation, 16
equivalent faults, 81
existential abstraction, 17
externally-cut circuit, 48

F

$\mathcal{F}$
  flow relation, 25
  mapping gates to functions, 50
failure state, 55, 70
false negative, 48, 65, 68
false positive, 65, 68, 75
fanin, 50, 52, 56, 63, 68
fanout, 50, 51, 63, 68
fault, 77
  activation, 88
  models, 80
  simulation, 91
feedback delay, 7
fixed point, 32, 35, 73
fork, 50
function, 16
  image, 16
functional correctness, 60
functional verification, 73
fundamental mode, 8

G

$\mathcal{G}$ (set of gates), 84
gate, 7
  collapsing, *see* collapsing
  combinational, 51
  complex, *see* complex gate
  excited, 52
  modeled with Petri nets, 52
  sequential, 51
  stable, 52
  switching, 51
gate delay, 8

H

hazard, 50, 61
hazard-free circuit with bounded delays, 9, 73
home state, 70
Huffman model, 8

I

identity relation, 16
idle component, 6
image, 16
image computation function, 52
  extension of, 54
inertial delay, 7, 50
initial marking, 23
initial state
  of a circuit, 50
  of an STG, 29
input pattern or vector, 77, 82
input stuck-at fault, 81, 93
isomorphic circuits, 69
`ite` (if-then-else) operator, 21

L

L0- to L4-live, 39–40
labeling functions, *see* $\lambda, \lambda_G, \lambda_I$
least fixed point, 32

least upper bound, 16, 90
literal, 16
liveness, 39, 70, 72
logic-level synthesis, 2
lub (least upper bound), 19

M

$[m\rangle$, 25
$m[t\rangle m'$, 25
m_0, 25
$[m_0\rangle$, 38
$[m_0\rangle$, 25, 32
$\mathcal{M}_{\mathcal{P}}$, 26
marking, 10, 23, 25, 53
  pre-unsafe, 39
member of a set, 14
memory element, *see* gate, sequential
memory elements, 61
metastability, 82
micropipelines, 9
minterm, 16, 26, 29
model checking, 48
Muller model, *see* speed-independent circuit
multiple fault, 81

N

n-variable function, 16
n-variable logic function, 26
negative excitation function, 52
next state function, 27, 84
non-confluence, 10, 80, 82, 83, 86, 87, 95
  not detected, 95
  practical detection, 87
non-transient cycle, 87
nondeterminism, 69
NPM$_t$, 28
NSM$_t$, 28

O

observability, 77
observational equivalence, 66, 68
one-to-one function, 16
operations with sets, 14
optimistic test, 4
ordered BDD, 20
oscillation, 10, 80, 82, 83, 86
  practical detection, 87
output stuck-at fault, 81, 93

P

$\mathcal{P}$
  set of places, 25
  set of primary inputs, 84
${}^\bullet p$, 25
$p^\bullet$, 25
p_i, 26
partial order, 16
path delay fault, 79
path in a state graph, 30
persistence, *see* transition or signal persistence
pessimistic test, 4
Petri net, 10, 23, 25, 38
  k-bounded, 26
  analysis, 24
  as circuit specification, 23, 53
  bounded, 25
  equivalent STG, 29
  example, 24
  modeling, 26
  reduction, 24, 33, 44
  safe, 26
place, 10
  graphical representation, 23
  marked, 23
  predecessor/successor, 23
positive excitation function, 52
potentially fireable transition, 39
power set, 15
  of circuit states, 51
predecessor place, 39
predicate, 13
preimage, 16
projection of a state graph, 30, 62, 68
proposition, 13
pure delay, 7

Q

quasi-delay-insensitive circuit, 9, 51

R

random test pattern generation, 92
reachable markings, 31, 38
reachable states, 38
reconvergent fanout, 82
recursive rule describing a set, 14
reduced BDD, 20
reflexive and transitive closure, 15
reject ratio, 4
relation, 15, 27

- Representation Theorem, *see* Stone's

S

- $s[t\rangle s'$, 29
- s^0
  - (initial STG state), 29
  - (initial circuit state), 50
- $[s^0\rangle$, 29, 38
- $\mathcal{S}_\mathcal{V}$, 29, 51
- safe Petri net, 39
- safeness, 38, 39
- scan-path, 78
- self-checking circuits, 78
- semi-modular, 59, 61, 66–68
  - circuit, 65
- semi-modularity, 56, 69
- sequential depth, 33, 35
- set, 14
  - of markings, 10
  - of reachable markings, 31
- sets of states, 51
- Signal Graph, 28,
  - *see also* STG
- signal persistence, 40
- Signal Transition Graph, *see* STG
- silent label, 29
- silent transition, 29
- specification validation, 10
  - goal, 23
- speed-independence, 54
- speed-independent circuit, 8, 50, 71
- state
  - disambiguation, 64
  - failure, *see* failure state
  - in asynchronous circuits, 50
  - in synchronous circuits, 5, 50
  - minterm, 51
  - notation, 50
  - of a Petri net, *see* marking
  - of an environment-circuit system, 53
  - of an STG, 29
- state differentiation, 89
- state explosion, 5, 25, 48, 90
- state graph, 30
  - as environment of complex gate, 69
  - as specification, 75
  - associated to a Petri net, 31
  - associated to an STG, 31
  - example, 55
  - of a circuit under test, 84
  - of a complex gate, 61
  - of an autonomous circuit, 50, 66
  - projection, *see* projection of
  - projection adding nondeterminism, 63
  - showing non-confluence, 83
- state justification, 88
- STG, 25, 28, 38
  - as circuit environment, 55
  - as circuit specification, 25, 38
  - as environment, 66
  - equivalent Petri net, 29
  - implementability, 38
  - modeling, 29
  - shorthand form, 29
  - state, 29
- Stone's theorem, 18, 26
- strong detectability, *see* detectability
- strongly-live circuit, 57
- stuck-at fault model, 81
- successor place, 39
- sum of minterms, 29
- switching algebra, 17
- symbolic BFS, 32
- symbolic model checking, 48
- synchronous circuits
  - pros and cons, 5
- synthesis, 1

T

- $\mathcal{T}$ (set of transitions), 25
- $^\bullet t$, 25
- $t^\bullet$, 25
- $\mathcal{T}_H$, 53
- $\mathcal{T}_I$, 53
- $\mathcal{T}_O$, 53
- T_{v*}, 29, 53
- T_{v+}, 29, 53
- T_{v-}, 29, 53
- temporal logic, 25, 48
- ternary algebra, 19
- ternary simulation, 90
- test, 77
- test cycle, 85
  - estimation, 86
- testing, 3
- testing asynchronous circuits, 10
- theorem proving, 48
- three-phase ATPG, 88
- toggling, 36
- token, 25
- topological image computation, 27
- trace theory, 48
- transient cycle, 87
- transition
  - fired infinitely often, 71
  - firing, 23, 27

  graphical representation, 23
  ordering, 36
transition function, 27
transition persistence, 40, 41
transition relation, 27, 40, 85
transitive closure, 15, 40

U

ULSI, iii
unbounded delay, 7, 60
unbounded gate delay, 50, 84
unfolding Petri nets, 48
universal abstraction, 17
universal set, 14
universe of discourse, 14
upper bound, 16

V

$\mathcal{V}$, 29, 49
$\mathcal{V}_G$, 50
$\mathcal{V}_H$, 50
$\mathcal{V}_I$, 50
v_i, v_k, 29, 52
$\mathcal{V}_O$, 50
valid test vector, 83
verification, 3
  flat, 57, 61
  goal, 49
  hierarchical, 58
  is not validation, 3
  of asynchronous circuit, 10
  of behavioral correctness, 61
  of functional correctness, 60
  of isomorphic circuits, 68
  results, 72
  scenario, 53
vertex, 16
VLSI, iii, 1, 9

W

wire, 7
wire delay, 8

Y

yield, 3