

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

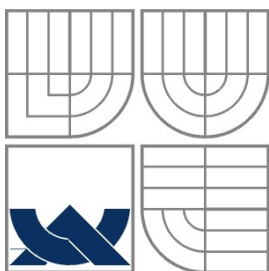
VERIFIKACE ASYNCHRONNÍCH
A PARAMETRICKÝCH NÁVRHŮ HARDWARE

DISERTAČNÍ PRÁCE
PHD THESIS

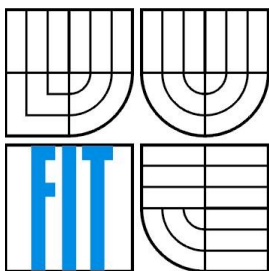
AUTOR PRÁCE
AUTHOR

ALEŠ SMRČKA

BRNO 2010



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

VERIFIKACE ASYNCHRONNÍCH
A PARAMETRICKÝCH NÁVRHŮ HARDWARE
VERIFICATION OF ASYNCHRONOUS AND PARAMETRIZED HARDWARE DESIGNS

DISERTAČNÍ PRÁCE
PHD THESIS

AUTOR PRÁCE
AUTHOR

Ing. ALEŠ SMRČKA

ŠKOLITEL
SUPERVISOR

Doc. Ing. TOMÁŠ VOJNAR, Ph.D.

BRNO 2010

Abstract

In this thesis, we introduce two original approaches to formal verification of hardware designs. In particular, we aim at model checking of circuits with multiple clocks and verification of parametrized hardware designs. Considering the former contribution, we introduce four methods which we use for modelling the clock domain crossing of a circuit. Models derived in such a way can then be model checked as usual while possible problems stemming from the synchronization within a circuit are implicitly covered. Four proposed ways of modelling a data transfer differ in their precision and the incurred verification cost. In the latter contribution, our proposed approach of verification is based on a translation of parametrized hardware designs to counter automata and on exploiting the recent advances achieved in the area of their automated formal verification. A parametrized hardware design translated to a counter automaton can be verified for all possible values of parameters at once.

Abstrakt

V disertační práci jsou prezentovány dva originální přístupy k formální verifikaci návrhů hardware. Konkrétně se věnujeme metodě model checking systémů s více hodinovými signály a verifikaci parametrických návrhů hardware. Co se týče prvního přínosu, v práci jsou představeny čtyři metody, které jsou použity pro modelování křížení časových domén digitálního obvodu. Na modelech, které jsou získány navrhovaným způsobem, může být aplikován model checking obvyklým způsobem, přičemž problémy plynoucí ze synchronizace dat digitálního obvodu zůstávají pokryty. Čtyři navrhované metody se liší v přesnosti a v nárocích na formální verifikaci. Další přínos disertační práce je založen na překladu parametrických návrhů hardware do čítačových automatů přičemž využívá současných úspěšných výsledků v oblasti jejich automatické formální verifikace. Parametrický návrh hardware přeložen do čítačového automatu potom může být jednorázově verifikován pro všechny možné hodnoty parametrů.

Key Words

Formal verification, modelling hardware design, clock domain crossing, parametrized hardware design, counter automata.

Klíčová slova

Formální verifikace, modelování návrhů hardware, křížení časových domén, parametrický návrh hardware, čítačové automaty.

Acknowledgement

First and foremost I would like to thank my supervisor doc. Tomáš Vojnar for his guidance, advice, and constructive criticism which have greatly helped me to progress towards the successful completion of this work. I would also like to thank prof. Milan Češka for his moral, professional, and financial support. I also have to express my deepest thanks to my wife Lucie for her infinite patience and endless support without which I would never finish my dissertation.

The work presented in this thesis was supported by the Czech Ministry of Education (project no. MSM 0021630528) and the Czech Science Foundation (project no. GP103/10/0306).

Contents

1	Introduction	1
1.1	Formal Analysis and Verification	2
1.2	Challenges in Formal Verification	5
1.3	Goal of the Thesis	6
1.4	Structure of the Thesis	7
2	Background	8
2.1	Hardware Design	8
2.1.1	Hardware Design in RTL	10
2.1.2	Syntax and Semantics of VHDL	10
2.1.3	Synthesisable Hardware	13
2.1.4	Transparent and Synchronous Mode	16
2.2	Model Checking Hardware Designs	18
2.2.1	Modelling Signal Propagation	19
2.2.2	Modelling Environment	20
3	Verifying Hardware Designs with Multiple Clocks	22
3.1	Related Work	22
3.2	Problems in Circuits with Multiple Clocks	23
3.2.1	Transient Behaviour	23
3.2.2	Clock Domain Crossing	24
3.3	Modelling Asynchronous Behaviour	27
3.3.1	The Basic Idea	28
3.3.2	Definitions	29
3.3.3	Extending All Critical Input Ports	31
3.3.4	Extending Critical Paths	42
3.3.5	Modelling with One-step Destabilizers	48
3.3.6	Delaying the Clock Domain Output	55
3.4	Algorithms of Finding Critical Ports	58
3.5	A Comparison of Modelling Methods	61

3.5.1	A Methodology of Formal Verification of Asynchronous Systems	65
4	Verifying Parametrized Hardware Designs via Counter Automata	67
4.1	Related Work	68
4.2	Counter Automata	68
4.3	Preprocessing Hardware for Translation to Counter Automata	69
4.3.1	The Considered Features of VHDL	69
4.3.2	Simplifying VHDL Code	75
4.3.3	A Normalization of Conditional Assignment Statements	79
4.3.4	Handling VHDL Integer Variables in Counter Automata	84
4.4	An Intermediate Behavioural Model	84
4.4.1	A Definition of the Intermediate Behavioural Model	84
4.4.2	Extracting Behavioural Rules from the Source Code	86
4.4.3	Adjustments of Behavioural Rules	87
4.5	Generating Counter Automata	89
4.5.1	Counters, Control Locations, and Initialization	89
4.5.2	The Transition Relation	91
4.6	Experiments	93
4.7	Summary of the Translation	96
5	Conclusion and Future Research	97
5.1	Verifying Clock Domain Crossings	97
5.1.1	Future Directions in Verifying CDC	98
5.2	Verifying Parametrized Systems	98
5.2.1	Future Research of Parametrized Systems	99
5.3	Other Work of the Author	99
5.3.1	A Methodology of High-level Modelling and Analysis of Hardware Designs	99
5.3.2	Proving CRC Algorithm Properties	100
A	An Example of the Implementation of the Destabilizer	109
B	Simple Handshake Synchronization Protocol	113
C	Examples of Abstractions of Bit-wise Operations in VHDL	116

Chapter 1

Introduction

It is widely agreed that our everyday life is increasingly supported by computer-based systems. The most important attribute of them, particularly when used in critical applications, is reliability. The complexity of modern computer systems, including the complexity of modern circuits, is rapidly rising. It has been observed that verification becomes a major bottleneck in hardware design development, up to 80% of the overall development cost or time [1, 2]. There are two approaches that are widely used in the verification phase of the development cycle of hardware systems: (i) simulation and testing, and (ii) formal analysis and verification.

Although *simulation* and *testing* can be seen as old-fashioned techniques, they are still quite effective, especially in the early stages of debugging. Both of them are used on two independent levels, namely, software simulation of HDL designs (the approach is very similar to testing of software) and testing of the real hardware, implemented using ASIC or FPGA technology. The testing of the real hardware benefits from the speed (testing is performed in real-time) and allows to cover the impact of the technology used to physically implement the circuit. HDL software simulation on the other hand allows the user to verify important parts of the design a component-wise in a very user-friendly manner before the circuit is actually physically assembled. Simulators provides excellent control over the analysis and the cost of verification via software simulation is low. One of the main attributes of using simulation-based verification, that allows an acceleration of the verification by it in a systematic way, is the so-called *coverage* measuring how much of the design has already been verified [82]. Simulation tools which are usually used by industrial companies are, e.g., Modelsim¹, Synopsys VCS², or, from the open-source area, GHDL³. There are also many coverage-based simulators that automatically generate

¹www.model.com

²www.synopsys.com/tools/verification/functionalverification/pages/vcs.aspx

³ghdl.free.fr

tests, monitors, and functional models—MPSim⁴, Specman⁵, and inFact⁶.

1.1 Formal Analysis and Verification

The second approach is *formal analysis and verification*. This thesis contributes to the field of formal verification, therefore we will discuss the technique in more details. The main idea of formal verification is to prove the functional correctness of a design instead of performing some random test cases, i.e., in a language of simulation experts, to exhaustively verify all possible test cases. Different techniques for the proof process have been proposed during last decades. We will not discuss all of them, instead, we will shortly introduce the techniques that various industrial companies and organizations proved to be effective in finding faults in a system.

Static analysis is a technique that analyses the source code of a design without actually executing it. In hardware development, static analysis is, e.g., often used to check that an HDL design meets preconditions of a correct use of particular components it is composed of. In particular, static analysis is often performed on components like counters, well-known synchronisers, standard bus controllers, etc., to check their proper connection to the rest of the circuit or to check the timing settings. Once the static analyser finds out that the some part of a circuit is connected against the predefined rules, it provides the user with the location within the HDL source code that causes the problem. We have to note that such a violation does not necessarily mean that the design is incorrect, in particular, special-purpose circuits often optimize the solution against standard rules. In a case of an optimized circuit with more than a million gates, static analysis often concludes with a great number of false warnings. On the other hand, the main advantage of the approach is that it can be performed fully automatically. Also the amount of time needed for the verification is short, no matter how large the verified circuit is (in comparison to other approaches). Examples of commercial static analysers include MCDV⁷, 0-In CDC⁸ [54], and WCET⁹.

Equivalence checking aims at proving the equivalence of two similar circuit descriptions. The description might be provided on different levels of abstraction, such as the register transfer level or gate-level, and in different formats (languages) or formalisms. To prove equivalence or inequivalence, the checker translates both designs to a unified internal format, and establishes the correspondence between them in a matching phase [5, 6]. In case of inequivalence, a counter-example is generated and provided to the user. Due to a need of two circuit descriptions, equivalence checkers,

⁴www.athdl.com/mpsim.html

⁵www.cadence.com/products/fv/enterprise_specman_elite/

⁶www.mentor.com/products/fv/infact/

⁷www.athdl.com/mcdv.html

⁸www.mentor.com/products/fv/0-in-cdc/

⁹www.rapitasystems.com/WCET-Static-Analysis

like FormalPro¹⁰, Conformal¹¹, and Formality¹², are suitable for the regression testing phase of the hardware development cycle, i.e., to check that no additional error is introduced in a new version of a circuit description.

When verifying a complex hardware design, it is often impossible to fully automate the verification process. As a result, the hardware design must be broken down into smaller pieces, either following the structure or partitioning the input data space [49] in a similar way as domain decomposition methods to solve a boundary value problems. Such a decomposition is made manually, thus there is a need to ensure that no mistakes are made in the partitioning process and that no verification conditions are forgotten. A proof of the correct partitioning can be achieved by *theorem proving*. Theorem proving is an approach of deducing mathematical theorems about the examined system. Such a reasoning is performed using computer programs called theorem provers [13] (like HOL Light¹³, Forte¹⁴ [11, 49, 12], or ACL2¹⁵ [50]), which are aware of the so-far deduced facts about the system and are equipped with inference rules allowing to deduce new facts from the already known ones. Most of theorem provers require a manual guidance from an expert to prove the desired properties of a given system, but there is also an effort of developing theorem provers to be fully automatic yielding the so-called decision procedures, which are then used within more complex theorem provers or also within approaches to formal verification (such as model checkers where decision procedures are important in automatic abstraction [52]).

Model checking [14] consists in verifying that a model of a system (or sometimes the system itself) satisfies a given property by exhaustively exploring the state space. A model checker automatically searches the state space of the system for behaviours that violate the given property. Since the number of states grows exponentially with the size of a system, model checking greatly suffers from state space explosion problem. However, even when the state space is too large or even infinite, applying model checking can still be useful—it may be unable to prove that the given property is satisfied, but it can still find a fault that violates the property. If a violation is found, a model checker provides the user with a counter-example showing the case when the property does not hold. The properties to be checked are often expressed by formulas of some temporal logic (e.g., CTL or LTL [14]) or in some specialized property specification language, typically inheriting temporal aspects from temporal logics (e.g., PSL [15] or FQL [16, 17]). Typical representatives of commercial model checkers of

¹⁰www.mentor.com/products/fv/formalpro/

¹¹www.cadence.com/products/ld/equivalence_checker

¹²www.synopsys.com/tools/verification/formalequivalence/

¹³www.cl.cam.ac.uk/~jrh13/hol-light/

¹⁴www.comlab.ox.ac.uk/tom.melham/res/forte.html

¹⁵userweb.cs.utexas.edu/users/moore/acl2/

hardware designs are Mentor's 0-In¹⁶ or RuleBase¹⁷ from IBM. Tools available for academic purposes are, e.g., Cadence SMV¹⁸ or NuSMV¹⁹.

Although model checkers are automatic tools, the cost of their integration into the verification phase of development can be very significant. This is due to several facts: (i) an environment of a system to be verified must be modelled manually, (ii) to cope with the state space explosion problem, some modification of the model of a system is usually required (e.g., abstracting away some details, restricting the scope of the analysis, etc.), and (iii) to properly encode properties of the system in formal temporal logic, a user with an expertise in the field of system specification is needed. The research of the last decades tries to lower these costs, in particular, to reduce a need of manual modifications of a model by automatically reducing the state space of the system being verified. Different techniques have proven to be useful in this task:

- *Symbolic methods* represent the state space of a system symbolically to avoid explicitly enumerating each individual variable value. The most popular representation is based on binary decision diagrams (BDDs) [7, 8, 9].
- *Bounded model checking* (BMC) [18] is model checking that considers counterexamples of a maximal length k . This restriction allows to unroll the model a given number of steps and to encode it via an instance of SAT problem (and to benefit from the advances in the technology of SAT solvers). Usually, the process of bounded model checking is performed in an iterative way while incrementing k until all possible violations are covered until some predefined limit is reached.
- *Partial order reduction* [14] analyses dependencies between concurrent processes to avoid exploring equivalent interleavings of independent operations. This is based on the principle that if two concurrent operations are independent, then it does not matter in which order they are executed. For a hardware design, partial order reduction contributes especially in verifying systems like controller units or components implementing some sort of communication protocol.
- *Abstraction methods* simplify a system before verifying it (we refer to a simplified system as an *abstract model*). The term abstraction represents a technique which generally consists in ignoring some aspects of the model of a system [47]. The most common technique is the abstraction by restriction [3],

¹⁶www.mentor.com/products/fv/0-in_fv/

¹⁷www.research.ibm.com/haifa/projects/verification/RB_Homepage/

¹⁸www.kenmcml.com/smv.html

¹⁹nusmv.irst.itc.it/

which is used in proving that the system satisfies the formula under some circumstances. Such an abstraction under-approximates the system by forbidding some behaviours wrt. given circumstances (e.g., the clock signal abstraction which we discuss in the next chapter). If a violation of the property is found in an abstract model, it is also violated in the original system. If the property is proven to be true in an abstract model, it is not guaranteed that it is true in all circumstances.

To keep the proof to be sound, the abstraction is also required to be sound—we say that an abstraction over-approximates the system behaviour. This could be achieved in two ways: (i) via abstraction based on state merging and (ii) abstraction on variables [3]. State merging reduces the number of states while creating more general states of the system. Abstraction on variables reduces or omits constraints over the system variables (variables which usually represent the “data” part of the system). The over-approximation is not complete (it introduces false alarms), thus if a violation of some property is detected on an abstract system, it should be verified in the original, not approximated, model. In case a false alarm is detected, it is desirable to be able to automatically refine the abstraction and repeat the verification—this yields the so called CEGAR loops [48, 19]. Examples of tools that implement CEGAR for verification of hardware include, e.g., VCEGAR [20] and BAT [27]. The most common used abstraction technique in CEGAR loop is *predicate abstraction* in which the abstract model is constructed using a given set of predicates on a data [22]. The technique has proven to be successful mainly in software domain [23, 24, 25, 26], but there is also an effort to apply predicate abstraction in hardware [21, 48, 20]. The results achieved in hardware are not so emphatic as in software mainly because the hardware is often designed in a low-level containing ubiquitous bit-wise operations. Bit-wise operations (such as overflow detection, data coding, etc.) do not fit to predicate abstraction which often aims at arithmetic and relational operations over real or integer numbers.

1.2 Challenges in Formal Verification

Even though model checking is successfully applied and has become the state-of-the-art in many design flows, still many problems remain. In this section, the list of some problems is given. Note that the list is not complete in the sense that all difficulties are covered, but we mention some important representatives including those that inspired the work presented in this thesis.

Complexity. Rising requirements on computer-based systems cause rising complexity of the designed systems. Model checking works best on the module level (for simple components like FIFO channels, simple bus or memory controllers, or logic units of microprocessors), but is unable to verify the entire system which integrates such

components into a large block. Many development workflows deal with this problem by combining several verification approaches, and the developers create their own methodology of verification, while a significant amount of work has to be done manually.

Diagnosis and automatic synthesis. After a fault has been identified and a model checker generates a counter-example, the next step is to identify the root cause of the discovered fault and remove it. Usually, the fix is performed manually, but there exist efforts to automatically synthesize the design from a formal specification directly [29, 30]. Unfortunately, such techniques are purpose specific and lack a generic usage.

Arithmetic. Industrial practice has shown that model checking has difficulties with floating-point arithmetic circuits, like multipliers or dividers [49, 50, 51]. Even though reasoning at a higher level of abstraction has gone through significant improvements (e.g., results achieved by using SMT paradigm over bit vectors [10]), it is often difficult to integrate these abstraction methods to a fully automatic verification process.

Multiple clocks. Model checking algorithms assume that the circuit is purely digital, but in a real circuit, the data is transferred via analog signals. A problem arises when verifying circuits with multiple clock domains, since if the model of such a circuit is digitalized in an improper way, errors due to delays of a real circuit may be missed. Hence, the model checking is unsound.

Generic designs. Modern HDL languages allow the developer to reuse the code describing a part of a circuit via the so called parametric description. Parametrization is widely used, e.g., when creating libraries of re-usable hardware components. If the circuit is described in a modular way, the parametric description is often related to *generic modules*. Usually, the number of different configurations of such modules is unbound. When verifying a generic module for all possible cases, the state space is infinite, but the current model checkers used in a hardware design flow work on a finite state space basis.

1.3 Goal of the Thesis

The goal of the thesis is to advance the current state of the art in two areas of hardware verification—namely, (a) model checking of designs with multiple clocks and (2) formal verification of parametrized designs.

As for what concerns dealing with designs with multiple clocks, we concentrate on clock domain crossing signals. We aim at proposing verification approaches capable of revealing errors related to clock domain crossing signals. These approaches should be fully automated, i.e., not requiring the users to understand. Current methods of modelling a system to be verified are unsuitable for model checking of clock

domain crossings as they hide the transient responses of a circuit. To deal with this, we provide four approaches to extend the model of a system with transient behaviour such that subsequent model checking is able to automatically reveal the synchronization faults.

The second aim of the thesis is to facilitate formal verification of parametrized designs. Currently, verification of generic modules includes verification of various concrete instances of the modules. In our approach, we were motivated by the recent advances in formal verification of a counter automata. The thesis provides the method how to transform a generic HDL design to a counter automaton to allow verification of all possible configurations of a design at once.

1.4 Structure of the Thesis

The rest of the thesis is organized as follows. Chapter 2 introduces the reader to the basics of hardware design HDL languages, and to the current techniques used for modelling such designs for their subsequent model checking.

Chapters 3 and 4 discuss both of the original contributions of the thesis. In both cases, we start with an explanation of the problem and the basic principles of the presented solution. We also provide the related work in the given area. Subsequently, a detailed description of the proposed techniques is provided. The chapters conclude with some remarks based on our experiments. In particular, Chapter 3 deals with verification of designs with multiple clocks and provides four approaches for model checking such designs. In Chapter 4, we discuss the area of verification of parametrized hardware designs via transformation of an HDL design to a counter automaton.

In Chapter 5, we conclude the thesis by a general summary of the discussed methods and provide directions of possible future research.

Moreover, to provide the reader with a better illustration of the contribution, some examples of applications of the introduced techniques are discussed in the appendices. In particular, Appendix A describes, on an example, an application of one of the methods proposed in the thesis. Appendix B contains the details of a synchronization protocol used in our experiments. Finally, Appendix C describes one of the ideas of future research on a simple example.

Chapter 2

Background

In the following, we will introduce the reader to the field of hardware development based on modern HDL languages. To better understand the problems described in the following chapters, the notion of a basic syntax of well-known hardware description language is presented, together with the common methods of modelling a circuit for the purpose of model checking.

2.1 Hardware Design

The term *hardware* is in general meant for any electronic circuit used in information technology (digital or analog). Since digital circuit design comprises most of electronic circuit design, in this section further, we consider a hardware as a *digital circuit*.

There are several ways of how to deal with hardware design, each of them uses different abstraction of a description. More abstract levels make the design very easy to understand, on the other hand, low-levels of hardware design allow the developers to control the optimization of the circuit (e.g., the number of used electronic components, low-power consumption, or the calculation speed of the circuit). Even though more and more of the effort is given to the area of the abstract hardware development (such as the algorithmic level of a description via ANSI C/C++/SystemC language), hardware developers still use less abstract level of a hardware description—*register transfer level* (RTL). In RTL, a hardware is designed at the level low enough for the possibility of optimizations and abstract enough for better understanding and for simple reuse of the code to create more complex circuit.

There are several languages for RTL abstraction, also known as *hardware description languages* (HDL), out of which the most widely used are VHDL and Verilog [28, 65]. A design specified in such a language is an input for hardware synthesis tool, hardware simulation, or formal verification (cf. Figure 2.1). A process called *synthesis* transforms a generic RTL description of a system to the gate-level descrip-

tion. The hardware description at this level is adjusted for a concrete electronic circuit capabilities, e.g., a set of available components of specified type of FPGA chip. The design described at a gate-level represents the logical structure of a hardware (an interconnection of hardware primitives). To obtain graphical representation of the circuit board (placed components and wires drawn between them) or the layout within FPGA, the process place&route must be performed.

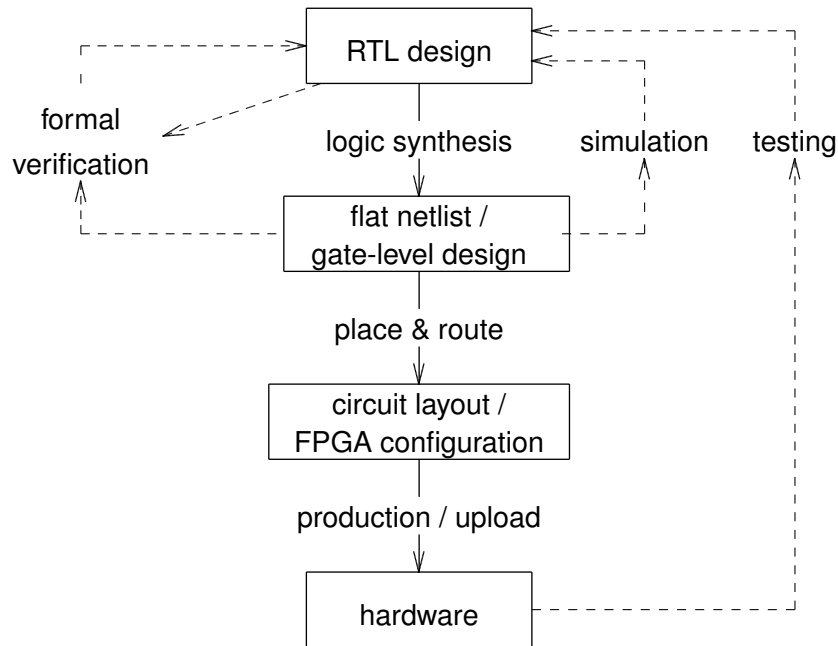


Figure 2.1: The integrated circuit design cycle aiming the development of a printed circuit board or a hardware based on programmable field array. The dashed parts of a flow represent validation stages which consist of formal verification, simulation, and testing. The term “simulation” is sometimes called as the *testing process*, and the testing itself as the *testing on a real hardware*.

Considering the validation back loop, there are several approaches of validation processes in HW design, each of them is suitable for different kind of checked properties. As for formal verification or simulation, the behavioural specification is verified. For the testing, the physical specification and the properties related to a real hardware is validated, such as thermal performance or the load on a real traffic. The logical choice of the design level at which we want to perform the formal verification is the logic structure of a design, i.e., the gate-level design¹. The testing on the other

¹For the case of formal verification of a generic hardware design, RTL design, instead of gate-level design, must be considered (we will discuss the problem in more details in Chapter 4).

hand must be performed on a real hardware. Such a design development cycle is well-known and used in most of the hardware projects.

2.1.1 Hardware Design in RTL

In register transfer level, the design is based on the connection of registers, logic gates, and data transfer between them. A register is a hardware element which is able to hold data—a bit or a bit-vector value. The source code of a design includes a declaration of communication channels of a bit or a bit-vector type (these will be the wires in a circuit), logic gates which provide a computation function, and the registers for holding the current state of a running circuit.

The most frequently used approach to a hardware design is a component-based design. In register transfer level, it is based on a definition of *components* which are connected together via communication channels. A component is a digital circuit which implements the specific operation (e.g., an arithmetic or a logic unit, queues, buffers, controllers, etc.) and communicates with its environment via input/output ports. The RTL design describes a hardware by encapsulating the basic functions into components, and by specifying a dataflow relation between them. Most of RTL languages offer the re-usability of a component description, thus the developer is able to use the code of a component several times without unnecessary reimplementing it. We call a template or a pattern of the component as an *entity* and the concrete representation of it (the instance of an entity) as the *component* itself.

In the present, there are two well-known and widely used languages of RTL design—VHDL and Verilog. Although VHDL and Verilog are different languages, their main expressive means are quite similar from our point of view of either verifying designs with multiple clocks or verifying parametrized designs. Since the VHDL language has more intuitive support for parametrized designs, we will discuss only the VHDL language.

2.1.2 Syntax and Semantics of VHDL

In this paper, we will not introduce full syntax of rich VHDL language, only the structure of VHDL source code will be described. The VHDL language is similar to other structural programming languages and the syntax is easy to understand even from examples. More detailed information can be found in VHDL tutorials or in the VHDL standard [73].

In VHDL, a hardware system is specified by an entity with input and output ports for communication with an environment. An example of a component is a USB stick with four I/O port pins or a computer adaptor card with ports of a PCIe slot. A more complex hardware system is described in a modular way using smaller and simpler components—e.g., a graphics processing unit can be designed by creating

and integrating the simplest components like binary data transformation units (adder, multiplier, binary-gray code translator) or data transfer controllers.

For the introduction to the VHDL syntax, we will not explain the whole VHDL, instead we will go through two simple entity examples (Figures 2.2 and 2.3). The first example implements the entity of the transformation of a natural number in a binary code to a number in the Gray code, the other is the example of a simple arithmetic unit for 64 bit numbers. Although the examples below is not complete wrt. syntax of VHDL, we rather put the emphasis to the demonstration purpose.

```
1  entity reg_gray is
2    generic(N: integer);
3    port (
4      bin: in bit_vector(1 to N);
5      asyn: in bit;
6      clk: in bit;
7      gray: out bit_vector(1 to N);
8    );
9  end entity adder;

10 architecture behavioural of adder is
11   signal bin2gray: bit_vector(1 to N);
12   begin

13     bin2gray(N) <= bin(N);
14     for i in 0 to N-1 loop
15       bin2gray(i) <= bin(i) xor bin(i+1);
16     end loop;

17     process(reset, clk)
18     begin
19       if (asyn = '1') then
20         gray <= bin2gray;
21       elsif (clk'event and clk = '1') then
22         gray <= bin2gray;
23       end if;
24     end process;

25 end architecture;
```

Figure 2.2: VHDL design of a component that transforms an integer number in two's complement code to the number in the Gray code.

In VHDL, an entity is described by a definition of its *interface* and its *body*. The interface of an entity defines *parameters* and input/output *ports*. For the reference to the examples, the interface of the entity in Figure 2.2 is defined at lines

```

1  entity arith_unit64 is
2    port (
3      opcode: in bit_vector(0 to 1);
4      a, b: in bit_vector(0 to 63);
5      c: out bit_vector(0 to 63)
6    );
7  end entity arith_unit64;
8  architecture my_unit of arith_unit64 is
9    signal a1, a2, n1, n2: bit_vector(0 to 63);
10 begin
11   add: entity adder
12         generic map(N => 64);
13         port map(i1 => a, i2 => b, o => a1);
14   neg_a: entity negation
15         generic map(N => 64);
16         port map(i => a, o => n1);
17   neg_b: entity negation
18         generic map(N => 64);
19         port map(i => b, o => n2);
20   sub: entity adder
21        generic map(N => 64);
22        port map(i1 => a, i2 => n2, o => a2);
23   with opcode select
24     c <= a1 when '00',
25         n1 when '01',
26         a2 when '10',
27         (others => '0') when others;
28 end architecture;

```

Figure 2.3: VHDL design of a simple 64 bit arithmetic unit. The structural description consists of four instances of `adder` and `negation` subcomponents. The dataflow description controls the output of a unit.

1–9, the interface of the entity in Figure 2.3 is defined at lines 1–7. The parameters allows a generic description of the entity (the entity in Figure 2.2 is a generic entity—it describes a Gray converter for any width of a bit-vector). The ports identify the communication channels between the entity and its environment. Communication channels within the entity are called *signals*. The body of an entity, in VHDL known as the *architecture* (lines 10–25 in the example 2.2 or lines 8–28 in the example 2.3), specifies the dataflow, the structural, and the behavioural description of the entity. Every type of a description specifies the relation between communication channels in a different way.

The *dataflow description* (lines 13–16 in Figure 2.2 or lines 23–27 in Figure 2.3) specifies the relation between ports or signals by predefined arithmetic, logic, or con-

ditional connectors. The syntax and the semantics of dataflow VHDL constructs and expression operators are shown in Table 2.1 and 2.2.

In *structural description* (Figure 2.3, lines 11–22), the hardware is designed in a hierarchical structure, i.e., the high-level unit is described via lower-level components, which are specified with interconnection of a circuit elements. We see all the units and primitive elements simply as *components*. A component is an instantiation of the entity. An instance of the entity is made if the mapping of signals to ports is provided. If the entity is a generic entity, the instantiation is only possible if all its parameters are assigned with particular values. For the reference, cf. Figure 2.3 where the generic entity `adder` is instantiated for width of 64 bits and its input/output ports (`i1`, `i2`, `o`) are connected to signals of the architecture (`a`, `b`, `a1`).

The *behavioural description* (Figure 2.2, lines 17–24) consists of processes running simultaneously. Each process specifies the behavioural characteristics of a hardware it represents. The behaviour of the process is described by sequential statements like assignment, loop cycle, or condition. We have to, however, note that sequential statements in VHDL processes have a different meaning than in typical programming languages—the sequence they are based on is not the execution sequence, but rather a sequence of preferences of how to proceed to the considered output values. We will discuss it a bit more in Chapter 4.

Since there is no way how to efficiently synthesize a hardware design from complex behavioural requirements, the behavioural and the dataflow description is widely used for a low-level description of parts of a system (e.g., logic functions, simple registers, counters), while the structural description is used for building more complex components or the entire system.

Generic entities

As we mentioned before, the entities of an RTL design can be *generic*. Such a language feature allows us to write the code with the same properties within one generic code—we can think of the entity as the template of a component. The entity is generic if the interface declaration (more precisely the `generic` section within the `entity` declaration) includes one or more parameters. Such parameters *statically* control structural, dataflow, or behavioural characteristics of an entity. Once the entity is ready to use, the instantiation of it is done by mapping of parameters to specific constants and input/output ports to appropriate signals. The component is then constructed specifically according to parameter values.

2.1.3 Synthesisable Hardware

In our models which we use for the formal verification, we come out from the RTL description and from the gate-level design. To properly model a circuit, we need to understand the internals of the logic synthesis process.

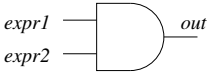
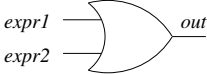

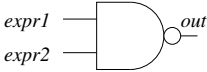


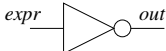
Syntax	Semantics	
<code>stmt ::= sig <= expr</code>	an assignment of the symbolic name <i>sig</i> to the output of the expression <i>expr</i> . The <i>sig</i> can be the name of a signal, the item of an array (e.g., <code>arrname(5)</code>), the field of a record (e.g., <code>recname.field</code>), or the combination (e.g., <code>rec.fld(0)</code>).	
<code>expr ::= signal</code>	$out \equiv$ value of <i>signal</i> (bit or bit-vector type)	
<code>expr ::= signal (i)</code>	$out \equiv$ value of <i>i</i> -th bit of bit-vector <i>signal</i>	
<code>expr ::= signal (i to j)</code>	$out \equiv$ value of slice, i.e., the bit-vector from <i>i</i> - <i>j</i> bits of the <i>signal</i> bit-vector)	
<code>expr ::= 0 1 '(0 1)+'</code>	$out \equiv$ bit value, or constant bit-vector	
<code>expr ::= expr1 and expr2</code>	$out \equiv expr1 \wedge expr2$	
<code>expr ::= expr1 or expr2</code>	$out \equiv expr1 \vee expr2$	
<code>expr ::= expr1 xor expr2</code>	$out \equiv \neg(expr1 = expr2)$	
<code>expr ::= expr1 nand expr2</code>	$out \equiv \neg(expr1 \wedge expr2)$	
<code>expr ::= expr1 nor expr2</code>	$out \equiv \neg(expr1 \vee expr2)$	
<code>expr ::= expr1 xnor expr2</code>	$out \equiv expr1 = expr2$	
<code>expr ::= not expr</code>	$out \equiv \neg expr$	

Table 2.1: VHDL constructs for dataflow description. Bitwise operations like `and`, `or`, or `not` over *bit-vectors* (*out*, *expr1* and *expr2* must have the same width) represent several bit-per-bit logic operations working in parallel, i.e., for the expression $a_1 \dots a_n \circ b_1 \dots b_n$, the resulting vector $c_1 \dots c_n$ is defined by $c_i = a_i \circ b_i$, $1 \leq i \leq n$.

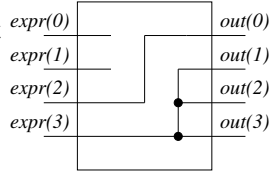
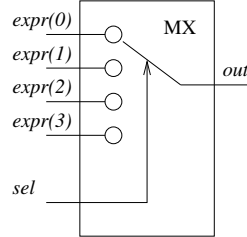
Syntax	Semantics
<pre> sll slr sal n sar rol ror expr ::= expr </pre>	<p>$out \equiv expr \circ n$, where $expr$ is a bit-vector, \circ is left/right shift, arithmetic shift, or rotate operation, and $n \geq 0$, e.g., $expr \text{ sar } 2$ (where $expr$ is 4-bit vector):</p> 
<pre> stmt ::= with sel select out <= expr0 when '00', expr1 when '01', expr2 when '10', expr3 when others </pre>	<p>choice one-from-many. Value of bit-vector sel selects which input value will be propagated to the <i>signal</i> (number of inputs = $2^{\text{width of } sel}$).</p> 
<pre> stmt ::= for i in range loop dataflow statements end loop </pre>	<p>The dataflow statements to be processed repeatedly. Within the dataflow statements, the loop parameter i is a constant of a base type of $range$.</p>

Table 2.2: Another VHDL constructs for dataflow description. Note that iterated loop cycle is often used for the dataflow description of a bit-vector (cf. Figure 2.2, lines 13–16).

The VHDL language apart from syntax described above also include the constructions for programming automatic test-benches, simulation, read/write operation over files, etc. These constructions are important for the hardware development process but insignificant for the logic synthesis process. The goal of the synthesis process is to create a hardware design ready to use for the implementation. For this purpose, there exist several fully-automated tools which translate the RTL design to the gate-level design, e.g., Design Compiler², Encounter RTL Compiler³, Leonardo Spectrum and Precision⁴, and Synplify Premier⁵. The design in a gate-level is represented by a structural description using real-life components only, such as registers, logic gates, multiplexers, address decoders, etc. The problem of the logic synthesis of an entity architecture is the problem of transforming every part of a VHDL description (dataflow, structural, and behavioural) to a gate-level structural description.

²www.synopsys.com/products/logic/design_compiler.html

³www.cadence.com/products/digital_ic/rtl_compiler/index.aspx

⁴www.mentor.com/products/fpga_pld/synthesis/

⁵www.synplicity.com/products/synplifypremier/index.html

For the *dataflow description*, the logic synthesis process is easy to understand. It is based on the translation of predefined operators to the gates as depicted in Table 2.1 and 2.2. Note that in the phase of synthesis, every parameter represents a constant value. Such an assignment (constant to a parameter) allows us to unfold every loop to obtain a dataflow description without any loop.

The *behavioural description* is not so easy to transform. The behavioural description is composed of so-called *processes* running in parallel. The process is described by sequential statements which represent the proper function. An important note here is that the process is not implemented as a sequence of operations performed one by one, instead, a logic synthesis tool analyses the code of the process and results with an appropriate circuit (hardware elements connected together) which represents the behaviour of the process. The synthesis of an arbitrary process is much more complicated than the synthesis of dataflow description. To synthesize an arbitrary behavioural description properly, complex techniques must be performed (SAT solvers or genetic algorithms could be involved) to find an appropriate replacement of the behavioural code in the form of gate components connected together. Such an approach of universal code analysis is not supported by logic synthesis tools. Instead, tools only support predefined templates of constructions of sequential statements. Such templates of behavioural code represent particular gate components. The only “programmable” parts of the behavioural description is a dataflow defined within the sequential statements (for instance, an assignment of logic expression to a variable), which is transformed as described above.

The *structural description* consists of entity instances. The synthesis of a structural description can be simply interpreted as a transformation of each of such instantiations. Note that a subcomponent (an entity instance) can also include behavioural, dataflow, or further structural descriptions. The synthesis which transforms the subcomponent to the desired form, translates its behavioural or dataflow descriptions as described above. For the case of structural description, the same procedure is performed recursively, but, in addition to that, some care must be taken to the signal names in order to preserve unambiguous signal identification in the whole hardware design. This can be achieved for example by a simple renaming of all subcomponent signals, more precisely by giving a unique name in the whole design to every subcomponent signal.

2.1.4 Transparent and Synchronous Mode

In the approaches of modelling the circuit, we also distinguish the behaviour of gates which work in the *transparent* or in the *synchronous mode*. These modes substantially influence the output of a gate.

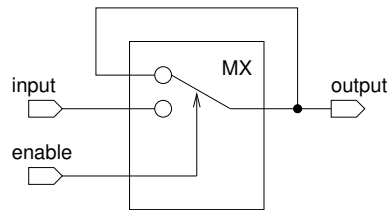
For example, let us have two gates connected in a cascade. If both gates work in the transparent mode and the input changes its value, the first gate instantly propagates the input to the output (the input of the second gate), and the same value

```

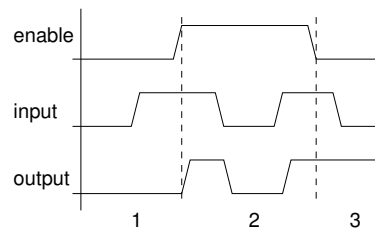
process(enable, input)
begin
  if enable = '1' then
    output <= input;
  end if;
end;

```

(a)



(b)



(c)

Figure 2.4: A latch is a circuit with the ability of holding a value. A latch works in a transparent mode and it is sensitive on the *value* of an enabling signal. The RTL design (a) consists of one process with conditional assignment (there is an implicit `else` section with the semantics `output <= output;`). The circuit (b) can be implemented via multiplexer with a back-loop (to hold a value, if the enabling signal is not set). The example of a timing diagram depicted in (c) includes three phases: holding a value, transparently setting a new value, and holding the last value from the time of a transparent mode.

propagation happens at the second gate. The result of the transparent mode is that the change of the input of the first gate instantly changes the value of the output of the second gate.⁶ The gate working in transparent mode only is called a *latch*—its design, implementation, and timing diagram is depicted in Figure 2.4.

Conversely, if both gates work in the synchronous mode (such gates are known as *flip-flops*, cf. Figure 2.5), they propagate their inputs to the output one step at a time—the change of the input values of the first gate changes its output after one clock period, but this still does not immediately influence the output of the second gate (its output is changed only after another clock period). Such a behaviour takes an advantage from the propagation delays such that the time, when a clock edge is detected and both gates are triggered for an action, is earlier than the input value of the first gate manages to propagate to the output. Finally, let us add that some gates

⁶According to physical latency called a *propagation delay* (which is a delay of a gate to propagate input signal to the output), the change is not so “instant”, but wrt. clock period of a circuit it is insignificant.

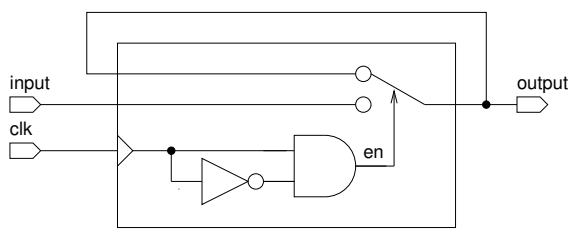
may be operated both in transparent and synchronous mode depending on some of their control inputs.

```

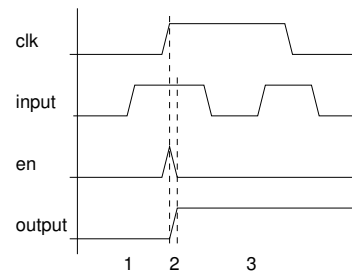
process(clk, input)
begin
  if clk'event and clk = '1' then
    output <= input;
  end if;
end;

```

(a)



(b)



(c)

Figure 2.5: A flip-flop is sensitive on the *edge* of the enabling signal, thus it is working in a synchronous mode. Such an enabling signal is called a *clock* and it is often used for controlling of every component in a hardware. The RTL design of a flip-flop (a) is similar to a latch (cf. Figure 2.4) except for the enabling condition includes the expression of an edge-detection. The scheme of a flip-flop (b) differs from the scheme of a latch in the semantically depicted edge detection by two logic gates (clk and not clk). The timing diagram (c) shows that signal output is set by a rising edge of signal clk in the second phase. The short peak of signal en (momentary enabled signal) is caused by propagation delays of the edge-detection circuit.

2.2 Model Checking Hardware Designs

Before a hardware design can be verified, one typically needs to create or automatically derive its model. Depending on circumstances, the model can be closer or farther from the actual system, but there are at least two issues that deserve a closer attention here: namely, modelling of the signal propagation in the circuits and modelling of its environment.

2.2.1 Modelling Signal Propagation

When designing a new digital hardware, developers consider the circuit under construction to be digital, but the real signals and logic operations are analog. In most cases, typically when dealing with synchronous hardware, the digital view on a circuit during the design is not a problem, since there exist automated methods and tools which are able to detect problems possibly caused by settings unachievable by real, analog circuits. But beyond these cases, e.g., when dealing with asynchronous circuits, there is no universal method how to model a system such that further verification of the model is able to reveal problems caused by wrong assumptions of an analog implementation of the digital design. Therefore, the modelling step is crucial—it is the task best performed by qualified engineers enlightened with a good grasp of both the physical reality and the mathematical models.

There exist methods that automatically model a digital circuit by an under-approximation, but, as we mentioned in Chapter 1.1, an under-approximation is not sound. In this section, we will discuss three of such modelling methods:

- *Modelling synchronous circuits* (as described in [14]) is the most common modelling method of digital circuits since most of them are synchronous, i.e., every gate in a system is controlled by the same clock signal. Verification of such models proves that the *logic* of the circuit function meets its specification, thus only the logic effect (not side effect) is relevant. In this case, one needs not to represent the clock signal in the states. Successive states are obtained by computing new values of all state variables from the current values, using the combinational logic of the circuit.
- *Zero-delay modelling* considers with the stable states of a circuit only. Briefly, the stable state of a circuit is the state that the circuit holds until an external event occurs. On the other hand, unstable states arise due to transition and propagation delays of real gates changing their stable states (cf. Figure 2.6)—we will discuss this in more detail in the next chapter. The zero-delay modelling is suitable for verification of circuits with multiple clock signals since the state of the circuit evaluates both common signals and the clock signals. Zero-delay modelling deals with binary values only. It is unable to model tri-state logic functions or the value of an input port of a component which is not connected. Further in the text, we call the model obtained by zero-delay modelling as the *zero-delay model*.
- *Modelling of transient behaviour* counts with propagation delays of gates. A logic signal in a real circuit is not restricted to binary values only, the modelling include states of a signal, e.g., rising edge, falling edge, metastable, strong/weak values, high impedance (in particular, IEEE standard [79] defines 7 possible values of each of signal). To create a model reflecting such states,

the electrical properties (the timing properties and voltage thresholds in particular) must be taken into account. The sequence of states in such a model reflects the real time-line of a circuit. We need to mention that such a modelling method is not suitable as an input for formal verification since non-binary values of every state variable of the system greatly increase the state explosion problem. Models taking into account the transient behaviour are suitable for a detailed analysis of the circuit via software simulation.

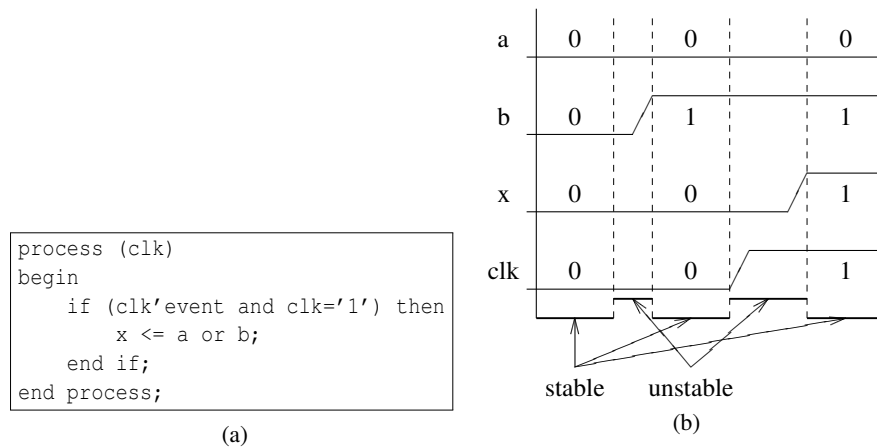


Figure 2.6: (a) The source code of a simple flip-flop component and (b) an example of a timed diagram of its behaviour illustrating the notion of stable and unstable states. Note that unstable states cover periods of time when one or more signals change their values.

2.2.2 Modelling Environment

When model checking a system, it is desirable to verify that the specification holds in all circumstances caused by the environment of a system. To cover all possible runs of a system, the specification of the environment should be such that it includes every situation that can actually happen. This can often be approximated by allowing a purely random behaviour of the environment. For example, in Cadence SMV, the random value of the signal `clk` in the next state of the system may be defined by the assignment statement:

```
next (clk) := {0,1};
```

The operator `next` on the left side of the assignment means that the statement defines the future/next value of the variable `clk`. The right side of the assignment

$\{0, 1\}$ represents a non-deterministic choice of values 0 and 1. Such a statement provides the variable `clk` with a random value in every step of a system execution.

The randomness of all clock signals allows the model to represent all possible frequency ratios and phase shifts between any two clocks. However, modelling all possible variations of clock signals is undesired in most cases since it does not represent a real scenario and complicates state space exploration (e.g., we should exclude from the model the behaviour in which the clock signal never changes). This can be easily achieved by using fairness assumptions expressed, e.g., in temporal logic formulae (for instance, $(\mathbf{G} \mathbf{F} \text{clk}) \wedge (\mathbf{G} \mathbf{F} \neg \text{clk})$ ⁷ specifies that the clock is alive, i.e., it is not possible to stop the function of a circuit).

In the thesis, we use the zero-delay modelling, in which clock signals are defined as external signals (their values are retrieved from the environment of the component being verified). In the case of a property which refers to a clock event, the event must be expressed explicitly: we use $(\neg \text{clk} \wedge \mathbf{X} \text{clk})$ for a rising edge of the signal *clk* and $(\text{clk} \wedge \mathbf{X} \neg \text{clk})$ for a falling edge. Such an explicit definition complicates specification of the properties to be checked, e.g., the property that ‘*the signal s is always set just for one clock period*’, looks like:

$$\mathbf{G} ((\neg \text{clk} \wedge (\mathbf{X} \text{clk}) \wedge \mathbf{X} s) \rightarrow \mathbf{X} (s \wedge \neg(\neg \text{clk} \wedge \mathbf{X} \text{clk})) \mathbf{U} (s \wedge \neg \text{clk} \wedge (\mathbf{X} \text{clk}) \wedge \mathbf{X} \neg s))$$

Just for the comparison with the model of a synchronous circuit where a sequence of states reflects a sequence of clock events, the same property looks much easier:

$$\mathbf{G} (s \rightarrow \mathbf{X} \neg s)$$

⁷We will not describe temporal logics, instead, we only briefly introduce some of temporal operators and refer the reader to other publications for detailed description [31, 14, 3, 4]. The linear-time temporal logic (LTL) understands every execution of the system as a sequences of states. An LTL formula representing a property of the system must hold in every execution of the system. LTL formula consists of atomic propositions connected with usual logic connectives ($\neg, \vee, \wedge, \rightarrow$) and the following temporal operators: temporal operator \mathbf{G} in the formula $\mathbf{G}f$ means that LTL formula f holds in the current state and in every future state, the formula $\mathbf{F}f$ means that f eventually holds some time in the future, formula $\mathbf{X}f$ means that f holds in the next state, and finally $f_1 \mathbf{U} f_2$ means, that f_1 holds in all subsequent states until f_2 holds. Computation tree logic (CTL) is, unlike LTL, a branching time logic in which all executions of a system may be represented with a computation tree (a tree of states starting in the initial state). The branching structure in the computation tree is described with CTL path quantifiers (\mathbf{A} and \mathbf{E}) which can occur only in pairs with temporal operators (and conversely, every temporal operator must be preceded by a path quantifier): $\mathbf{AX}, \mathbf{EG}, \mathbf{A}(f_1 \mathbf{U} f_2)$, etc., where $\mathbf{A}\pi$ (or $\mathbf{E}\pi$) means that, in a particular state, through all executions (or some executions respectively) starting at that state, π must hold.

Chapter 3

Verifying Hardware Designs with Multiple Clocks

During the process of hardware design, synchronization problems are sometimes underestimated (especially when optimizing the circuit) which results to an invalid circuit. In this chapter, we aim at verification of data transfer between two mutually asynchronous clock domains. We start with a summary of related work in this area and we provide a description of possible synchronization problems. Then, we aim at eliminating the need of manual verification of synchronization solutions by providing an automatic method for deriving a model of the transient behaviour that can manifest in a given design. A design composed with such a model can then be model checked as usual while possible problems stemming from the synchronization are implicitly covered. Four different ways of modelling the transient behaviour, differing in their precision and the incurred verification cost, are in particular proposed. Two of these methods were originally published in [92]. The two other ones have not yet been published, they will be submitted for publication soon.

3.1 Related Work

There have appeared multiple papers in the area of verification of asynchronous systems or systems with multiple clock domains in the past two decades. We will reference the reader to the main papers only.

Solutions of a proper data transfer within the circuit or between separate circuits (so-called synchronization problem) begin with the analysis of the impact of the transition delay [44]. Several solutions have been proposed: the simple one-wire solution via a two-flip-flop synchronizer [32], a dual-clock FIFO synchronization channel [33], or predictive synchronizers [37] for periodic clock domains. In [39] and [34], possible problems of a synchronization are discussed and verification methods of some of presented problems are introduced. The methods include several areas

of verification using static analysis [35, 36], using simulation-based verification with System Verilog assertions [55], or using automatic formal verification [38].

We have to note that all of these verification methods try to cover well-known methods of synchronization, but there is no universal verification method for an arbitrary synchronization. In [39], synchronization problems and their verification are discussed. However, for the verification of a handshake protocol, the authors admit that “*the check involves intense user intervention, because automatic analysis of signals involved in a handshake protocol is not trivial.*”. This chapter proposes several automatic methods of verification of synchronization subsystems, *including transaction-based protocols.*

3.2 Problems in Circuits with Multiple Clocks

Data transfer within a circuit controlled by multiple clocks requires a synchronization. Failures due to incorrect synchronization are difficult to detect and developing verification methods in this area requires in-depth understanding of so-called *clock domain crossing*. Before addressing clock domain crossing, we introduce the principle of data transfer within digital circuit.

3.2.1 Transient Behaviour

As digital circuits are implemented using electric circuits working in analog domain, one must consider how digital data are transferred within a circuit. Here we aim at transient behaviour, i.e., how the binary signals propagate through the circuit.

Transition Time

When dealing with the transient behaviour, we have to take into account what happens when a signal changes between two adjacent states, e.g., on a rising edge (the signal changes from the low level to the high level), or on a falling edge (from the high level to the low level). In real circuits, changes of a signal take a non-zero time. The amount of time that the output of a logical circuit requires to change its state is called the *transition time*, which is dependent on parameters of circuits such as transistor technology, capacitive loads, and voltage thresholds.

Figure 3.1 shows transition times, i.e. the rise time t_r and the fall time t_f of a signal in a logical circuit. The values V_L and V_H indicate the voltages for the LOW and HIGH values of a signal, V_{Hmin} and V_{Lmax} are the maximum and minimal thresholds for which the signal is recognized at LOW and HIGH values. If the signal level is between V_{Hmin} and V_{Lmax} , it is understood as metastable, i.e., has an undefined value.

For further analysis, we take into account that a signal, when changing its value,

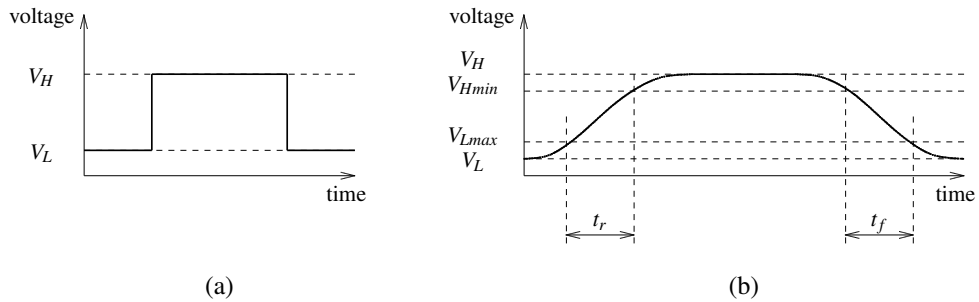


Figure 3.1: An example of (a) an ideal and (b) the real transition graph of a signal.

has an undefined value for a short period of time t_{rf} ¹.

Propagation Delay

If we focus on logic gates, we have to consider another type of delay which can cause synchronization problems. If the input of a logic gate changes its value so that this event will invoke the change of the output signal, it takes some time to propagate the input to the output (see Figure 3.2). Such a delay is called *propagation delay* representing the time needed for an input to be propagated through the gate to the output. It is usual that different types of logic gates have different propagation delays, whose value depends on the voltage settings, temperature of the environment, and used technology of the circuit being manufactured. For example, some combinational logic gates have smaller propagation delay than t_{rf} , which is smaller than propagation delay of sequential gates [40, 41].

The consequence of propagation delay in the system is that, for a short period of time, the signal values may lead to *inconsistent state*. For example, the `xor` gate behaviour depicted in Figure 3.2b shows that, in a time between the events e_1 and $e_1 + \Delta_{xor}$, there can be the configuration such that $in_1 = 0$, $in_2 = 1$, and $out = 0$, which is inconsistent with the `xor` logic function.

3.2.2 Clock Domain Crossing

The design of a multiple clocks system can be decomposed to several *clock domains*. A clock domain is a set of flip-flops and logic gates which are controlled by the same synchronization signal (clock signal).

The problematic point of data transfer in a multiple clocks system is the so-called *clock domain crossing (CDC)*. CDC denotes a situation when information from one

¹As for most of circuits, $t_r \cong t_f$, we use t_{rf} to represent the time of both the rising and the falling edge of a signal.

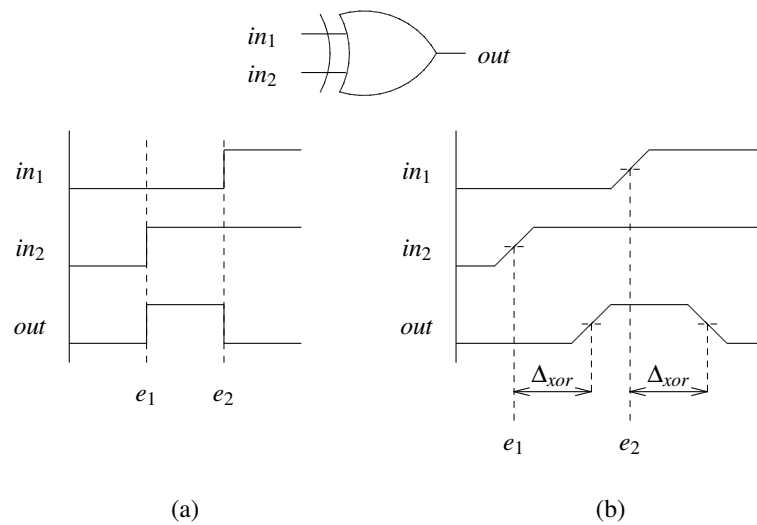


Figure 3.2: An ideal and approximated timing diagram of a `xor` logic gate. The bevelled edges represent the transition time of the rising and the falling edge of a signal. In the ideal model (a), the output signal reflects the event e_1 or e_2 in no time, while in the real system (b), the response takes the time Δ_{xor} .

part of a design controlled by one synchronization clock (the source clock domain) is transferred to a part controlled by a different clock signal (the destination clock domain). As these two parts of a design are physically connected by wires and possibly by logic gates, such a connection “crosses” two clock domains. Until there are no constraints about phase shift of clock signals (i.e., the source clock is multiple-times slower or faster than the destination clock and one clock depends on another), the time between two clock signals fire the execution of flip-flops can be arbitrary small. In such a case, the synchronization problem appears when both of the clock signals fire very closely causing that the source clock domain starts changing its flip-flop outputs which are read, at that exact moment, by logic gates of the destination clock domain. The destination clock domain can then obtain metastable or inconsistent values of transferred signals.

Metastability

The main problem of synchronous data transfer is *metastability*. If a logic gate reads input signal which, at the moment, has a metastable value, i.e., it is neither true nor false, the metastability of the input can be propagated to the gate outputs, thus they are also considered as metastable. Under some circumstances (design unsafe wrt. metastability, a bad choice of the clock frequency, setup and hold time re-

quirements of flip-flops or latches are not met, etc.), a metastable value can spread through the whole system. There have been proposed several methods how to keep metastability away from a design. It is widely considered that the two-stage synchronization circuit [28] is a general solution preventing metastability from propagating through a system. However, there exist automatic verification methods which check that the design uses proper metastability prevention [34, 35]. For instance, advanced EDA tools [81, 45, 46] warn the developer that designed system has a problem with metastability. That is why, in the following, we assume that the designs we are dealing with are metastable proof and we do not take metastability into account any more. In other words, we assume that the gate input always receives the signal value stabilized to either 0 or 1.

Inconsistent State

Even though the design is metastable proof, there is a risk of inconsistency in a given system. If the source clock domain sends a stable binary value, which is read by the destination clock domain as metastable and then stabilized with a proper stabilization mechanism, the resulting stable value can be inconsistent with the sending value (cf. Figure 3.3). For this purpose, the designer must enhance the design with a proper synchronization method².

Considering synchronous systems, if the data consistency specification is provided (in some sort of safety properties of a given system), a simple way how to verify inconsistency of data transfer is model checking the zero-delay model of a system. For an asynchronous system, such a method is unsound as the zero-delay model does not cover the transitional behaviour between stable states and hides the possibility of receiving random (inconsistent) values. Currently, to the best of our knowledge, the designers of RTL systems with multiple clocks solve the synchronization problem by using well-known synchronizers (FIFO queues, Gray-coded counter values, ...), or by introducing a new data transfer protocol (e.g., a hand-shake protocol) fit for their particular design. The advantage of the use of a well-known synchronizer is that the verification of a proper synchronization can be done by simple static analysis, i.e., by checking that the synchronizer is connected properly between two clock domains. Although this approach is suitable for the most of designs communicating via a standard interface (e.g., a computer bus), for designs where the size and efficiency are a top priority, applying universal synchronizers may be a too heavy solution. Instead, a solution with some kind of a hand-shake protocol could be more practical. The only and main disadvantage is that it involves a lot of non-trivial user interven-

²On the contrary, this is not the case if the information transferred between two clock domains is a one-bit value only. In such a case, either (i) the signal does not change, a metastability does not occur, thus the destination receives stable value, or (ii) the one-bit value is changing and the destination randomly chooses between 0 and 1, the destination receives the previous or the new stable signal value (which is in conformity with the source and the destination clock phase shift consequence).

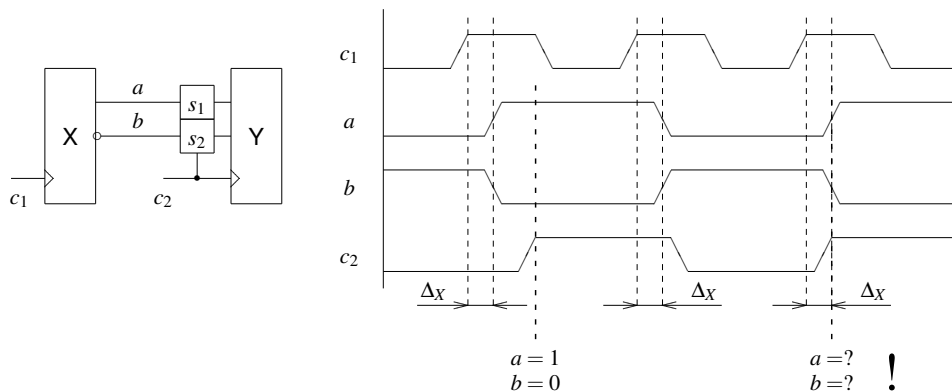


Figure 3.3: An example of inconsistency caused by clock domain crossing (signals c_1 and c_2 are clocks, a and b are data transfer signals, and Δ_X is the propagation delay of gate X). Let us say that the circuit is in a consistent state if $\uparrow c_2 \rightarrow a = \neg b$ holds (the expression $\uparrow c$ means that signal c is at the end of a rising edge). A problem occurs if the clock signal c_2 fires sooner at time Δ_X after the time of firing of c_1 . In such a case, the values of a and b are randomly chosen which violates the state consistency assumption.

tion [39] because transmit-receive protocols may differ wrt. their particular use in a system. Since such solutions may be quite tricky, a proper verification is desirable.

3.3 Modelling Asynchronous Behaviour

In this section, we provide four methods how to model a system in a way which exposes possible data inconsistency flaws caused by a wrong synchronization in the system. Each of the methods focuses on modelling the rising and falling edges of binary signals enabling detection of possible flaws in the synchronization. We model the system using the zero-delay abstraction described in Section 2.2 and enhance the model at places which may produce inconsistent states. Further in the chapter, we call such an enhancements as the *model extension*.

All four proposed methods differ mainly in their efficiency. The first method (extending all critical input ports) is quite precise in a matter of modelling possible synchronization faults, but it rapidly increases the size of a model. The second method (extending critical signal paths) tries to lower these costs with an over-approximation of asynchronous data transfer. Both of methods has been published in [92], the next two methods are new. The third method (modelling with one-step destabilizer) is inspired by its predecessors but model inconsistent states in a more efficient way. The fourth method (extending clock domain outputs) aims at false alarms caused by

a coarse over-approximation of presented methods and provides a more refined model of a given system.

The description of each method starts with an explanation of principles of a given extension. Then the implementation details are discussed and a justification of a solution is provided. Concerning the implementation, we provide a solution for Cadence SMV model checker [56]. We provide a source code of a model extension, in which we reference to the current and the next values of a state variable. Since most of BDD-based model checkers provide such a feature, it is also possible to apply our approach in other verification engines, such as NuSMV or Rulebase.

3.3.1 The Basic Idea

The basic idea of all methods is in generating a random phase to the propagation of a signal through a critical signal path whenever a signal changes. The fact that we make the propagation random stems from the reality where a signal does not sharply change from 0 to 1 (or vice versa) but goes through some rising (or falling) edge. When the signal is sensed by some gate on such an edge, one cannot predict its value. Each method of extending a model introduce the random phase on different parts of a circuit and with different lengths of phase. In general, the proposed approach is based on two assumptions:

1. clock signals are random all the time (cf. Section 2.2.2), and
2. model checking analyses all possible behaviours including all clock phase shifts.

Theoretically, in an extended model, when a source clock domain changes a signal propagating to another clock domain, a model checker examines all possible scenarios which include the following three cases:

1. The destination clock domain reads a signal before its change,
2. a signal is sensed within a random phase representing a rising or a falling edge of a signal, and
3. a signal is read after it is stabilized.

If a signal is sensed by the destination clock domain before or after the random phase, such situations clearly justify stable behaviour of a system. Reading of a random value simulates the case that the destination clock domain reads the signal exactly at time when the signal is changing.

3.3.2 Definitions

A Hardware Design

In order to precisely define the notion of gates, signals, ports, and critical signal paths, we view a particular RTL *hardware design* in an abstract way as a tuple $H = (S, C, P, G, M)$ where:

- S is a finite set of *signals*.
- C is a finite set of *clock signals*, $C \cap S = \emptyset$. In order to obtain a more regular description, we introduce a special clock signal $\perp \notin C \cup S$ that we associate with combinational gates. We denote $C_\perp = C \cup \{\perp\}$.
- P is a finite set of *gate ports*.
- $G \subseteq C_\perp \times 2^P \times 2^P$ is a finite set of *gates* (combinational logic gates, flip-flops, or latches). A gate—we use the notation $g = (c, I, O) \in G$ in the text below—is represented as a tuple consisting of its clock signal c (which is \perp for combinational gates), a set of input ports $I \subseteq P$, and a set of output ports $O \subseteq P$ such that $I \cap O = \emptyset$.
- $M : P \rightarrow S$ is a *signal interconnection function*.

A hardware design $H = (S, C, P, G, M)$ is *valid* iff:

1. no port is shared by two or more gates, i.e., $\forall g_1 = (c_1, I_1, O_1), g_2 = (c_2, I_2, O_2) \in G, g_1 \neq g_2 : (I_1 \cup O_1) \cap (I_2 \cup O_2) = \emptyset$,
2. every signal is set by one output only, i.e., for a signal $s \in S$ and gates $g_1 = (c_1, I_1, O_1), g_2 = (c_2, I_2, O_2) \in G$, if output ports $p_1 \in O_1$ and $p_2 \in O_2$, $M(p_1) = M(p_2) = s$, then $p_1 = p_2$, and
3. every port is connected with some signal, i.e., M is a total function.

Further in the text, we implicitly suppose that a hardware design is valid.

Signal Path

A *signal path* π is a string of gates and their I/O ports interleaved by signals connecting the ports (\dots , gate, output port, signal, input port, gate, \dots). Formally, for a hardware design $H = (S, C, P, G, M)$, a *signal path*

$$\pi = \langle g_1 o_1 s_1 i_2 g_2 o_2 s_2 i_3 g_3 \dots g_{n-1} o_{n-1} s_{n-1} i_n g_n \rangle \text{ of length } n > 1$$

is a connected sequence of gates, ports, and signals such that

$$\forall j \in \{1, \dots, n-1\} : g_j = (c_j, I_j, O_j) \in G \wedge g_{j+1} = (c_{j+1}, I_{j+1}, O_{j+1}) \in G \wedge$$

$$o_j \in O_j \wedge i_{j+1} \in I_{j+1} \wedge s_j \in S \wedge (o_j, s_j) \in M \wedge (i_{j+1}, s_j) \in M$$

For a signal path $\pi = \langle g_1 o_1 s_1 i_2 g_2 o_2 s_2 i_3 g_3 \dots g_{n-1} o_{n-1} s_{n-1} i_n g_n \rangle$, we denote:

- $\Gamma(\pi) = \{g_1, \dots, g_n\}$ the set of all the gates which appear in the signal path,
- $i(\pi) = \{i_2, \dots, i_n\}$ the set of all the input ports in the signal path,
- $i_i(\pi) = i_2$ the source port of the signal path,
- $i_o(\pi) = i_n$ the destination port which the signal path leads to,
- $o(\pi) = \{o_2, \dots, o_{n-1}\}$ the set of all the output ports in the signal path,
- $\Sigma(\pi) = \{s_1, \dots, s_{n-1}\}$ the set of all signals in π ,
- $\gamma_i(\pi) = g_1$ the input gate,
- $\gamma_o(\pi) = g_n$ the output gate,
- $\sigma_i(\pi) = s_1$ the input signal,
- $\sigma_o(\pi) = s_{n-1}$ the output signal.

We also denote $\Pi(H)$ the set of all signal paths of H .

A Clock Domain

We partition the set of gates G of a hardware design $H = (S, C, P, G, M)$ into subsets called *clock domains* that contain gates driven by the same clock signal. For $c \in C_\perp$, the clock domain is $D_c = G \cap (\{c\} \times 2^P \times 2^P)$. A special clock domain D_\perp stands for the set of gates which are driven by no clock signal (obviously, it is the set of all combinational logic gates).

Critical Input Ports

The set ι_c of *input ports critical wrt. a domain D_c* , $c \in C$, is the set of input ports which occur on signal paths that start in a different domain and lead to D_c and that are connected to the gates in D_c via combinational gates only. Equivalently, for a domain D_c , critical input ports are all the input ports on the signal paths that start by a sequential gate lying in a different clock domain and lead via combinational gates to a sequential gate in D_c . Formally, for a hardware design $H = (S, C, P, G, M)$,

$$\iota_c = \{p \in i(\pi) \mid \pi \in \Pi(H), \Gamma(\pi) = \{g_1, \dots, g_n\}, n > 1 : g_1, \dots, g_{n-1} \notin D_c \wedge g_n \in D_c\}$$

The set $\iota(H)$ of *critical input ports of H* is then simply the union of all the input ports critical wrt. the particular domains of H , i.e., $\iota(H) = \bigcup_{c \in C} \iota_c$.

A Critical Signal Path

Finally, a *critical signal path* of length $n > 1$ in a hardware design $H = (S, C, P, G, M)$ is a signal path

$$\rho = \langle g_1 o_1 s_1 i_2 g_2 o_2 s_2 i_3 g_3 \dots g_{n-1} o_{n-1} s_{n-1} i_n g_n \rangle \in \Pi(H)$$

that consists of critical input ports, i.e., $\forall j \in \{2, \dots, n-1\} : i_j \in \mathfrak{t}(H)$, and goes from one clock domain to another one, i.e.,

$$g_1 \in D_{c_1}, g_2 \dots g_{n-1} \in D_{\perp}, g_n \in D_{c_2}, c_1 \neq c_2, c_1 \neq \perp, c_2 \neq \perp.$$

We denote $\rho(H)$ the set of all critical signal paths in H . In the sections below, we will also use $[\rho]$ as the set of critical signal paths of sharing the same destination port, i.e.,

$$[\rho] = \{\rho' \mid \rho' \in \rho(H), i_o(\rho') = i_o(\rho)\}.$$

3.3.3 Extending All Critical Input Ports

We now discuss in detail the approach when we extend the zero-delay behaviour of every connection of a signal and an input port on critical signal path to make its value random for a *single verification step* (a step in the execution of a model) whenever there is a change of the stable value. We have to note that since we introduce a one step delay before every critical input port, there is a specific case of accumulated delay in signal path intersection which requires further consideration. In the following, we present the basic idea, we then optimize it for a BDD-based model checker. Finally, we describe the procedure of the enhancement of the model with proposed additional behaviour, we examine signal path intersection, and provide a justification of a construction.

Delayed Input

The principle of the transformation is the following. To model the impact of rising and falling edges in a critical input port, we replace every connection of a signal to a critical input port by a new gate. The new gate delays the original signal value before propagating it to the original input—we call the new gate a *delayed input* (cf. gate Δ in Figure 3.5a put between s and i_2). Formally, we change critical signal path $\rho = \langle g_1 o_1 s_1 i_2 g_2 o_2 s_2 i_3 g_3 \dots g_{n-1} o_{n-1} s_{n-1} i_n g_n \rangle \in \Pi(H)$, for $n > 1$, by inserting the extra delayed input Δ before every critical input port of $i(\rho)$ such that new critical signal path will be

$$\rho' = \langle g_1 o_1 s_1 i_{\Delta_2} \Delta_2 o_{\Delta_2} s_{\Delta_2} i_2 g_2 \dots g_{n-1} o_{n-1} s_{n-1} i_{\Delta_n} \Delta_n o_{\Delta_n} s_{\Delta_n} i_n g_n \rangle.$$

That is, every connection of signal and input port $\dots s_{j-1} i_j \dots$ in signal path ρ is transformed to $\dots s_{j-1} i_{\Delta_j} \Delta_j o_{\Delta_j} s_{\Delta_j} i_j \dots$ in signal path ρ' where, for $j \in \{2, \dots, n\}$, i_{Δ_j}

and o_{Δ_j} are the input and output ports of the Δ_j delayed input gate, and s_{Δ_j} is the new signal which connects the new gate with the original input port i_j .

We introduce the behaviour of Δ_j by the finite automaton in Figure 3.4. The values which label the arcs represent values of signal s_{j-1} , the control states identify values of output $s_{\Delta_{j-1}}$.

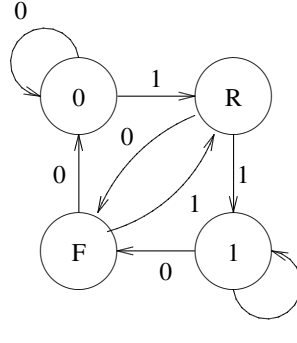


Figure 3.4: The finite automaton describing a delayed input value with rising and falling edges of signals.

Further, as digital gates are designed to handle only 0 and 1 values, we model the R and F values as a random choice between 0 and 1 in the final model (which we denote as the so-called x-value in the following). As we intend to verify the data inconsistency using exhaustive state space search, it is sufficient to implement the x-value as a non-deterministic choice between 0 and 1—non-determinism is implicitly handled such that every possible value is considered when computing the set of reachable states.

We illustrate the use of a delayed input on an example shown in Figure 3.5a where the delay is inserted behind an inverter with a non-temporal function $o = \neg a$. Here, when the delayed input is $s_{\Delta} = 0$ and the original signal value changes to $s = 1$, the automaton goes to R, and the delayed input becomes R (i.e. “rising”). Only then, provided the original signal value does not change, it transfers to 1. Similarly, for a change of the original signal value from 1 to 0, the delayed input goes from 1 to F (“falling”) and then changes to 0.

As an example, let us consider two gates in different clock domains D_{c_1} and D_{c_2} connected via signals a and b (cf. the circuit in Figure 3.6a). Let us say that the circuit is in consistent state if $x \neq y$ holds when firing the clock domain D_{c_2} ³. An example of the behaviour of the system under the zero-delayed model is depicted in timing diagram in Figure 3.6a. We can see that the inconsistencies which arise at time t_{err} are

³The event of firing the clock domain D_c by a rising edge of the clock c is expressed by a state in which the LTL formula $\neg c \wedge \mathbf{X} c$ holds.

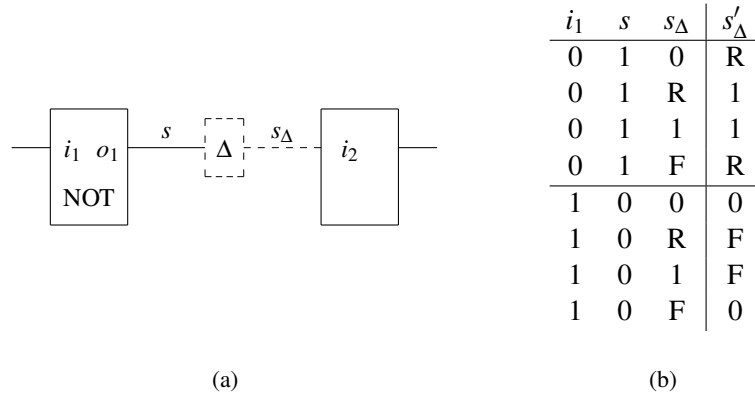


Figure 3.5: (a) An example of the delayed input (dashed part of an image) connected between the original connection of an inverter and a critical input port i_2 (i.e., $M(i_2) = s$), (b) the transition table of the delayed input (the signal value s represents the input tape of the finite automaton in Figure 3.4, the signal value s_Δ represents the current control state and s'_Δ the future control state).

hidden here. On the other hand, if we add the delayed input gates before every critical input, the inconsistencies appears (see x-values depicted as crosses in Figure 3.6b).

An Optimization of the Delayed Input

Concerning the state space explosion problem of an exhaustive state space exploration, one of the important properties of a model is the number of its state variables. An implementation of the finite automaton of a delayed input described above consists of four control states, which means that every delayed input adds to a model two bit variables representing its current state. Here, we will show that only one bit variable is necessary to obtain the same behaviour. This fact comes from two properties: (i) in a BDD-based model checker (which we intend to use), when specifying the future value of a state variable, it is possible to refer to the current and the future values of state variables, and (ii) only 2 state values are considered on the output of the delayed input gate (either 0 or 1).

The rising or the falling edge of a signal is zero-delay modelled as an immediate transition from 0 to 1 or vice versa. In such a case, when the current value at the input of a delayed input gate differs from the next value, a delayed input provides a non-deterministic choice of binary values in its output. Otherwise, when the current and the future value of the input do not change, a delayed input holds the same value on the output.

Such a behaviour is represented by the finite automaton in Figure 3.7. The label

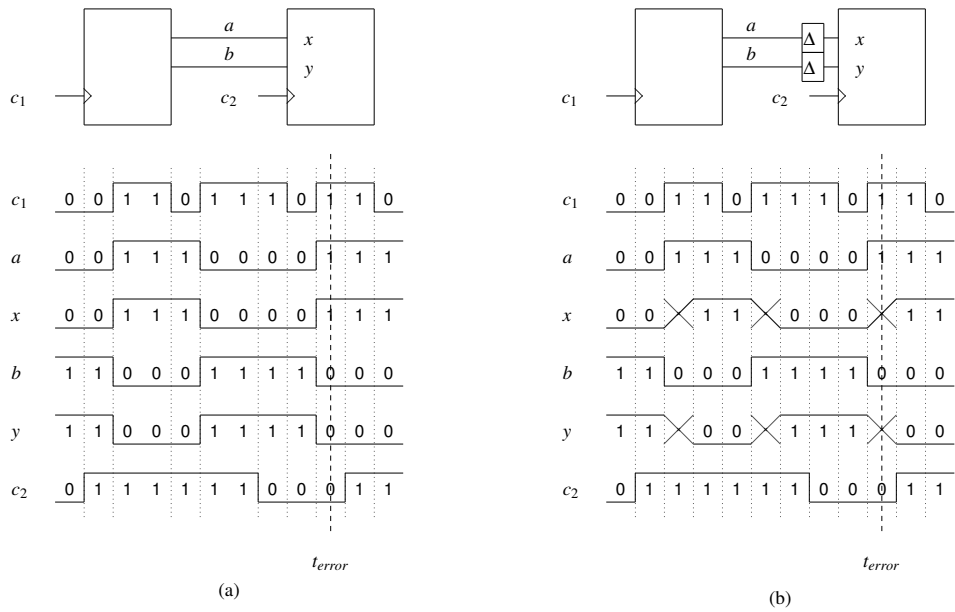


Figure 3.6: An example of two gates in different clock domains D_{c_1} and D_{c_2} as (a) a zero-delayed model, and (b) model extended with delayed inputs.

$s_j \neq s'_j$ represents the case that the input changes in its current and the next state. Similarly, the label $s_j = s'_j$ represents the case that the input holds its value. Control states define the values of the output of an optimized delayed input. Here, the state s_j produces the value which corresponds to the value at the input, and the state x produces the x-value—a non-deterministic choice between 0 and 1.

To prove that the automaton in Figure 3.7 is equivalent to basic, not optimized automaton in Figure 3.4, we construct the transition table of two subsequent transitions. The construction is performed as follows: for every two subsequent transitions $q_0 \xrightarrow{l_1} q_1$ (the transition from the state q_0 to the state q_1 labelled with l_1) and $q_1 \xrightarrow{l_2} q_2$

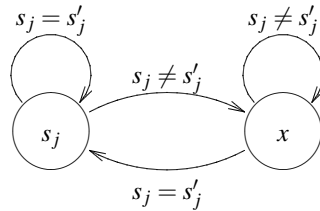


Figure 3.7: The automaton of an optimized delayed input gate.

row no.	q_0	l_1	l_2	q_2
1	0	0	0	0
2	0	0	1	R
3	0	1	0	F
4	0	1	1	1
5	R	0	0	0
6	R	0	1	R
7	R	1	0	F
8	R	1	1	1

row no.	q_0	l_1	l_2	q_2
9	1	0	0	0
10	1	0	1	R
11	1	1	0	F
12	1	1	1	1
13	F	0	0	0
14	F	0	1	R
15	F	1	0	F
16	F	1	1	1

(a) (b)

Table 3.1: A transition table of a delayed input. Each row depicts two transitions at once, where q_0 and q_2 are control states connected with two subsequent transitions with labels l_1 and l_2 .

in automaton in Figure 3.4, we create a row in a transition table which contains q_0 , l_1 , l_2 , and q_2 . In the original automaton, there are four states, each of them has two incoming and two outgoing transitions, that is, there are 16 couples of two subsequent transitions. Table 3.1 depicts the constructed transition table.

In the transition table, for every q_0 , if both l_1 and l_2 differ, the control state q_2 is either R or F (rows 2, 3, 6, 7, 10, 11, 14, and 15), which is identified in the automaton in Figure 3.7 by the control state x with incoming transitions $s_j \neq s'_j$ where $s_j = l_1$ and $s'_j = l_2$. Similarly, if both l_1 and l_2 have the same value, the resulting state produces the value of the label l_1 or l_2 (rows 1, 4, 5, 8, 9, 13, and 16), which is identified by the control state s_j with incoming transitions $s_j = s'_j$.

Implementing Delayed Inputs in Cadence SMV

We now have a look at how to apply the method of extending a model in Cadence SMV, thus the following text discusses implementation details of proposed model extension. We use the feature of several synthesizers being able to create the zero-delayed model of an RTL design, we then extend such a model on all critical input ports with delayed inputs defined above. The syntax of the source codes below conforms to Cadence SMV input language, which specification can be found in [56] or in [57].

We start with a description of extending combinational logic gates. Recall the zero-delayed model of a given design described in Section 2.2. In such a model, circuits of combinational logic gates (NOT, AND, OR, NAND, NOR, etc.) are translated to

SMV simply in the form of logic expressions. Now, we have to:

1. implement a *delayed input module* which delays a signal before propagating it to another gate input,
2. transform logic expressions of combinational logic of a circuit to a structural description, and
3. instantiate delayed input modules (one instance for each critical input port).

In the following, we will describe these steps in detail.

1. Implementing a delayed input module. In SMV, we create a common module `delayed_input` which will be instantiated several times. For comparison purposes, we start with the implementation of the basic, not optimized proposal of the delayed input (cf. Figure 3.8).

```
1 module delay_input(signal, delayed)
2 {
3     input signal : boolean;
4     output delayed : boolean;
5     state : {q0, qR, q1, qF};
6
7     -- an initialization phase
8     init(state) := signal ? q1 : q0;
9     -- state control
10    next(state) := switch(state) {
11        q0 : next(signal) ? qR : q0;
12        qR : next(signal) ? q1 : qF;
13        q1 : next(signal) ? q1 : qF;
14        qF : next(signal) ? qR : q0;
15    };
16
17    -- output generation
18    delayed := switch(state) {
19        q0 : 0;
20        qR : {0,1};
21        q1 : 1;
22        qF : {0,1};
23    };
24 }
```

Figure 3.8: A source code of a delayed input module in Cadence SMV.

By comparing the implementation with the automaton in Figure 3.4 on page 32, we believe that the construction is straightforward. The module has a binary input

```

1 module delay_input(signal, delayed)
2 {
3     input signal : boolean;
4     output delayed : boolean;
5
6     -- an initialization phase
7     init(delayed) := signal;
8     if (signal = next(signal))
9         -- the signal value is stable
10        next(delayed) := signal;
11    else
12        -- value changes, output the x-value
13        next(delayed) := {0, 1};
14 }

```

Figure 3.9: A source code of an optimized delayed input module.

(signal), a binary output (delayed), and an internal state. The variable `state` is a state variable⁴, which increases the size of a model being verified. To make the picture complete, we need to define the initial value of the control state, which we simply choose as the stable value of the module input. Note that the initial state of the model strongly depends on the specification used within the verified property and it is commonly assumed that the initial state represents some consistent and stable state.

In the following, the optimized delayed input implementation is presented. Note that there is no internal variable representing the current control state. Instead, the one-bit output variable `delayed` is a state variable delaying the input value. If we look at the automaton in Figure 3.7, the state x is represented as non-deterministic choice of output values (line 13) under the condition that the current input signal value differs from the next value (line 8). Otherwise, the output is defined as stable (control state $\{0, 1\}$), that is, the value is either 0 or 1 depending on the value of input signal (line 10).

2. Transforming logic expression. We transform each logic expression based on a signal appearing in a critical signal path to an appropriate structural description using modules. We create a module for every basic logic operator used within the expressions. Such a logic operator is implemented in the destination architecture as one gate only (typically NOT, AND, and OR). We then transform the logic expression to a structural description such that every use of an operator is substituted with

⁴State variables in SMV can be simply identified by their definition (the left side of an assignment) via the `init()` or the `next()` operators (cf. line 8 and 10).

```

1 module and(out, in1, in2)
2 {
3     input in1 : boolean;
4     input in2 : boolean;
5     output out : boolean;
6
7     out := in1 & in2;
8 }

```

Figure 3.10: A SMV module representing an AND gate with two inputs.

an instance of binary logic module. An example of an AND module is provided in Figure 3.10, other basic logic expression modules are very similar and conceptually differ at line 7 only.

In Figure 3.11, we give an example of transformation of an assignment of the value of a logic expression to a structural description. We believe that such a modification is straightforward and thus further explanation is unnecessary.

<pre> 1 z : boolean; 2 w : boolean; 3 x : boolean; 4 y : boolean; 5 6 z := w ~(x & y); </pre>	→	<pre> 1 z : boolean; 2 w : boolean; 3 x : boolean; 4 y : boolean; 5 s1 : boolean; 6 s2 : boolean; 7 8 g1 : and(s1, x, y); 9 g2 : not(s2, s1); 10 g3 : or(z, w, s2); </pre>
---	---	--

Figure 3.11: Transforming logic expressions to a structural description.

3. Instantiation of delayed input modules. At last, we create instances of delayed input modules between critical input ports and signals connected to them. For any gates $g_1, g_2 \in G$ and a signal s connecting some output o of g_1 with some critical input port i of g_2 —meaning that there is a signal path $g_1 o s i g_2 \in \Pi(H)$, we create one instance of the delayed input module Δ and a declaration of an auxiliary variable s_Δ . We then reconnect signals of the circuit in a way that conforms to signal path $g_1 o s i_\Delta \Delta o_\Delta s_\Delta i g_2$ where i_Δ and o_Δ are input and output ports of the delayed input gate.

To continue with our example in Figure 3.11, let us say that there is a critical signal path $g_1 o_1 s_1 i_2 g_2 o_2 s_2 i_3 g_3$ where i_2 and i_3 are critical input ports. In particular, the second port of the `not` module instance g_2 and the third port of the `or` module

instance g_3 are critical. The model extension contains the signal path

$$g_1 o_1 s_1 i_{\Delta_1} \Delta_1 o_{\Delta_1} s_{\Delta_1} i_2 g_2 o_2 s_2 i_{\Delta_2} \Delta_2 o_{\Delta_2} s_{\Delta_2} i_3 g_3,$$

so we create two instances of the delayed input module and two auxiliary variables representing signals s_{Δ_1} and s_{Δ_2} (cf. Figure 3.12).

```

1  z : boolean;
2  w : boolean;
3  x : boolean;
4  y : boolean;
5  s1 : boolean;
6  s2 : boolean;
7  s_delta1 : boolean;           -- a new signal
8  s_delta2 : boolean;           -- a new signal
9
10 g1      : and(s1, x, y);
11 delta1  : delayed_input(s1, s_delta1); -- delaying s1
12 g2      : not(s2, s_delta1);         -- s1 is replaced by s_delta1
13 delta2  : delayed_input(s2, s_delta2); -- delaying s2
14 g3      : or(z, w, s_delta2);         -- t1 is replaced by s_delta2

```

Figure 3.12: An example of an instantiation of delayed modules.

Signal Path Intersection

A problem with the above extension by delayed inputs arises when there exist two signal paths sharing one or more gate inputs and only one of the paths is a critical path. Formally, there exist signal paths

$$\pi_1 = \langle g_{1,1} o_{1,1} s_{1,1} i_{1,2} \dots o_{1,n_1-1} s_{1,n_1-1} i_{1,n_1} g_{1,n_1} \rangle \in \Pi(H)$$

and

$$\pi_2 = \langle g_{2,1} o_{2,1} s_{2,1} i_{2,2} \dots o_{2,n_2-1} s_{2,n_2-1} i_{2,n_2} g_{2,n_2} \rangle \in \Pi(H)$$

of a hardware design $H = (S, C, P, G, M)$ for $n_1, n_2 > 1$ and some domains

$$c_1, c_2, c_3 \in C, c_1 \neq c_3, c_2 \neq c_3$$

such that

$$g_{1,1}, g_{1,n_1} \in D_{c_1}, g_{2,1} \in D_{c_2}, g_{2,n_2} \in D_{c_3} \\ \forall j \in \{1, 2\} \forall k \in \{2, \dots, n_j - 1\} : g_{j,k} \in D_{\perp}, \text{ and } i(\pi_1) \cap i(\pi_2) \neq \emptyset. \quad (3.1)$$

In such a situation, delayed inputs in critical signal path π_2 would influence the behaviour of π_1 which is supposed to be zero-delayed. A problem arises due

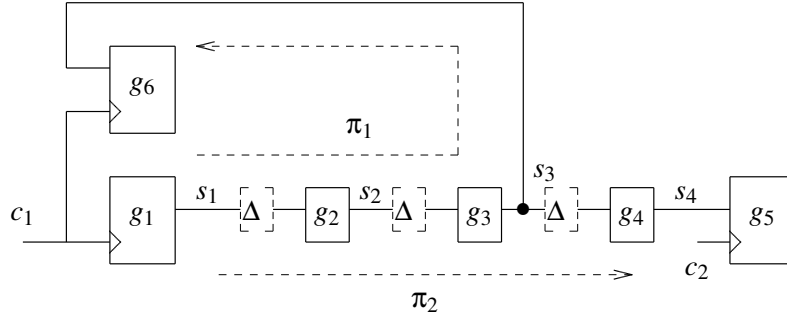


Figure 3.13: An illustration of signal path intersection. Critical input ports to be extended with delayed inputs are depicted using Δ gate.

to unconstrained clock signals (cf. Section 3.3.1 where we assume a clock to be random all the time) since it is possible for the clock domain D_{c_1} to be fired again after two verification steps. Signals originating from and leading to the same clock domain should be stable at firing event. If a delay introduced by delayed inputs in π_1 accumulates in more than one verification step, i.e., $|i(\pi_1) \cap i(\pi_2)| > 1$, this could lead to generating false alarms.

To prevent this false behaviour, before adding the delaying gates as described above, we pre-process the circuit by modifying π_1 into a new signal path π'_1 which bypass signal path π_2 and preserves the original behaviour. In such a case, a new signal path π'_1 will not contain critical input ports, thus it will be zero-delayed. The signal path π_1 is constructed by replacing shared parts of signal paths π_1 and π_2 (the gates shared by both signal paths, with their input and output ports, and signals connected to them) by newly created (duplicated) gates, ports, and signals with the same behaviour as the original. As the modification is simple, we explain the problem on an example depicted in Figure 3.14 based on the example in Figure 3.13.

More precisely, to modify π_1 into π'_1 , (i) we create new variables representing signals set by combinational gates shared by both signal paths (cf. Figure 3.14, signals s'_2 and s'_3 represent signals s_2 and s_3 set by g_2 and g_3), (ii) we duplicate such gates by newly created instances (gates g'_2 and g'_3 conforming to g_2 and g_3 —in SMV, such a duplication can be done by duplicating the instance statement of the original gate and by renaming newly created module instance), and (iii) we reconnect shared signals (s_2 and s_3) connected to duplicated gates by previously created signals (s'_2 and s'_3). Moreover, the duplication of a gate may violate the validity of a model (a valid model is defined on page 29) by duplicating an output port connection to some signal (in particular, to a signal not included in π_1 or π_2). In such a case, we regain the validity by substituting duplicated signal connection with an output port connection to newly created dummy variable.

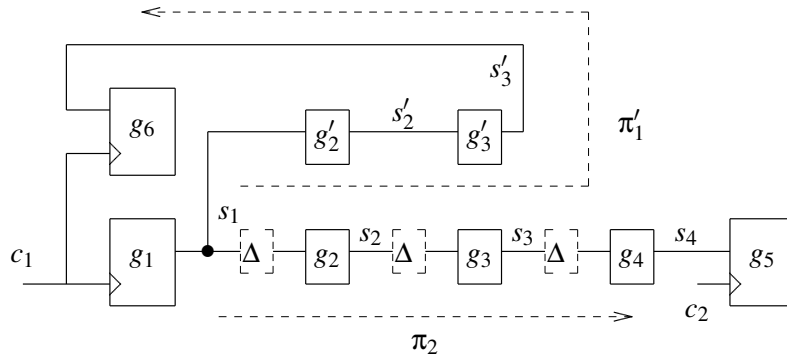


Figure 3.14: An illustration of gate and signal duplication.

For a hardware design H and for all $\pi_1, \pi_2 \in \Pi(H)$ which intersect each other, we modify π_1 to π'_1 as described above to get a new hardware design in which π'_1 and π_2 do not intersect. We repeat this step on a new hardware design until we receive a hardware design which does not contain signal path intersection conforming to formula 3.1. Performing the model extension with delayed inputs on such a design does not introduce false alarms caused by signal path intersection.

A Justification of the Construction.

The *extension of all critical input ports* is justified by our assumption that common RTL development tools are used to check that in all single clock domains, all signals have always enough time to stabilize before being sensed by sequential gates. Moreover, we assume that input and output signals of the entire checked design will be used, by another design, within the same clock domain as the gates which generate/consume these signals respectively.

As for the extension of critical inputs, the modification makes their values *random for a single verification step* if the changed input would lead to a value change. If this change is permanent, the extension is clearly justified because when the signal is rising from 0 to 1 or falling from 1 to 0, it can be sensed in an unpredictable way by the adjacent logic. On the other hand, when there is no change on the output, no extension is needed. An interesting situation is when there is a change on the output, but a temporary one only (the so-called hazard)—i.e. there is a rising and immediately a falling edge (or vice versa) [58]. In such a case, our approach introduces two random phases, which is again justified for most common-life cases as it is difficult to guarantee that the generated glitch in the signal would never be sensed (in any case, a design that would depend on this, would not be very clean).

The above justification is, however, valid only from the point of view of monitoring a single signal. When we look at *reachable combinations of multiple signal*

values, the length of the random phase (the phase with x-value(s)) is also important. We make it uniformly one verification step long which requires some further considerations. In fact, in general, such an approach can introduce an under-approximation or over-approximation though it does not happen in most practical situations (and it can be statically checked whether such a situation arises or not). In particular, such cases can arise *when the involved gates significantly differ in their delays*.

Let us consider the case of *two critical paths with a different length* (a generalization to more such paths is straightforward). Suppose we have two critical paths ρ_1, ρ_2 of lengths n_1, n_2 such that $n_1 < n_2$. If the accumulated delay of the gates in ρ_1 is smaller than in ρ_2 , clearly the output of ρ_1 will stabilize before the output of ρ_2 , which corresponds to our model. On the other hand, if the accumulated delay of the gates in ρ_1 is equal or greater than in ρ_2 , we need to be able to obtain the desirable combination of two x-values at the ends of both paths. That is, in our model, the signal at the output of the longer signal path ρ_2 must provide either the new stabilized value, or the x-value, at the time when ρ_1 is also providing the new stabilized value, or the x-value before its stabilization. Considering that the delayed input providing the x-value, which is a random choice between 0 and 1, also represents zero or one step of propagation delay⁵ (cf. Figure 3.15), the model without regard to the length of signal path ρ_1 simulates all possible scenarios of propagation delays on ρ_1 (0 up to n_1 steps of delay). More precisely, in our model, there is possible a case in which the shorter and slower signal path ρ_1 provides the old stabilized value (delayed by one or more of its delayed inputs) and the longer and faster signal path ρ_2 propagates the new stabilized value through all of its delayed inputs (which do not delay the signal). Moreover, there is a case, in which both critical signal paths delay the signal just for a single step (e.g., for both signal paths, all but one delayed input model the propagation as zero-delay). Thus, there is a time in which ρ_1 and ρ_2 are providing the x-value on the outputs.

3.3.4 Extending Critical Paths

The previous section provides a method of modelling the progressive delay of a critical signal propagation (and of the associated random phases when its value is changing) via an extension of every critical input. This method is rather precise but may cause a significant state-space explosion due to the number of newly introduced state variables. Below, we try to avoid this explosion by introducing a less precise, approximate model that can, according to our experience, be still sufficient in some practical

⁵For example, let us assume some signal has the waveform 011 (in the first state, the signal has value 0, then it rises to 1, and holds it in the next state). Its delayed form will have the waveform 0x1 where x represents two possible values caused by randomized delayed input, i.e., the delayed input generates two possible waveforms, either 011 or 001. The first waveform represents a zero-delayed signal propagation. The latter waveform simulates a one step delay of the signal value (1 appears after the second state, 0 remains a little longer). Similarly, the same representation of x-value holds for a falling edge.

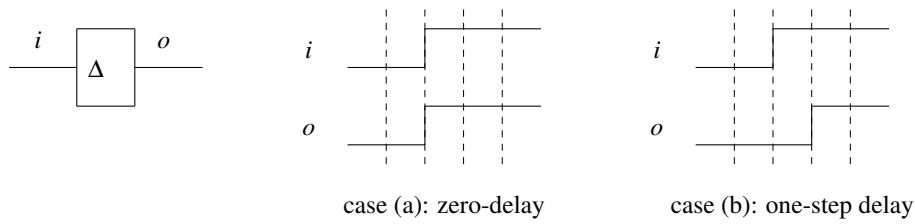


Figure 3.15: An illustration of two traces which are both possible when model checking a design consisting the delayed input Δ . Note that when n delayed inputs are connected in a cascade (they belongs to a critical signal path of the length n), model checking considers 2^n traces with different scenarios of a signal propagation including propagation delays of zero and n steps.

cases. In this approach, we do not extend every single critical input port, but instead, we put a special new gate called a *destabilizer* on every output of a critical signal path.

As a basis which we try to over-approximate in the new approach, let us summarize how the process of stabilization of a signal $\sigma_o(\rho)$ in a critical signal path ρ looks like in the previous approach when we extend every critical input by the delaying and randomising phase. In that case, a critical input port can be viewed as a generator of stable and unstable values. If more critical input ports are sequentially connected (they all appear in the same critical signal path ρ), the unstable values are propagated through all critical input ports on the path. A new value of the signal $\sigma_i(\rho)$ thus influences signal $\sigma_o(\rho)$ after the delay equal to the sum of delays of all delayed inputs on ρ . When $\sigma_i(\rho)$ changes its value, it can cause a temporary instability—the adjacent gates switch their output value through rising or falling edges and the undefined value is propagated to further gates. Due to modelling the delay of one gate as one step, it takes L steps to influence $\sigma_o(\rho)$ by $\sigma_i(\rho)$ where $L = |i(\rho)|$, i.e., unstable values of signal $\sigma_o(\rho)$ can occur in at most L steps. We discuss the justification of such an approach below.

A Destabilizer

The principle of the approximate approach we propose is to replace the progressive generation of unstable signals by having a single new gate called a *destabilizer* which will generate all possible combinations of x-values of a signal for a period of L steps. The destabilizer will be connected to the destination port of a critical signal path ($i_o(\rho)$) where x-values can become visible. We create one destabilizer for every set of critical signal paths having the same destination port. The destabilizer starts to generate x-values if one of the input signals of considered signal paths changes its value.

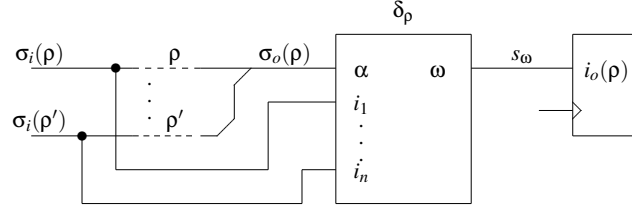


Figure 3.16: A schematic connection of a destabilizer to critical signal paths.

Formally, for a design $H = (S, C, P, G, M)$, a *destabilizer over critical signal paths* $[\rho]^6$, $\rho \in \rho(H)$ is a gate

$$\delta_\rho = (\perp, \{i_1, \dots, i_n, \alpha\}, \{\omega\})$$

where $I' = \{i_1, \dots, i_n, \alpha, \omega\}$ is a set of new ports, i.e., $I' \cap P = \emptyset$, for which $n = |[\rho]|$ is the number of critical signal paths with the same output signal. Apart from introducing δ_ρ , we have to insert the destabilizer to the hardware design H . We reconnect the original output signal $\sigma_o(\rho)$ of the given critical path ρ (and of the adjoining paths $[\rho]$) to input port α . To properly connect the output of the destabilizer with destination port $i_o(\rho)$, we create a new signal s' and connect it to ports ω and $i_o(\rho)$. Finally, since the destabilizer needs to monitor the input signals of $[\rho]$, we connect input signals of ρ and of the adjoining paths to i_1, \dots, i_n (cf. Figure 3.16).

In particular, by adding the destabilizer for a signal path ρ to the hardware design $H = (S, C, P, G, M)$, we change H to $H' = (S', C, P', G', M')$ where:

- $S' = S \cup \{s_\omega\}$, $s_\omega \notin S$, is the set of signals containing new signal connecting the destabilizer with the destination port of ρ ,
- $P' = P \cup I_\rho \cup \{\alpha, \omega\}$, $I_\rho = \{i_{\rho'} \mid \rho' \in [\rho]\}$, $(I_\rho \cup \{\alpha, \omega\}) \cap P = \emptyset$, is the set of ports with the new ports of δ_ρ ,
- $G' = G \cup \{\delta_\rho\}$ is the set of gates with the destabilizer
 $\delta_\rho = (\perp, I_\delta \cup \{\alpha\}, \{\omega\}) \notin G$, and
- $M' = (M \setminus \{(i_o(\rho), \sigma_o(\rho))\}) \cup M_\delta$ is the port–signal mapping where $(i_o(\rho), \sigma_o(\rho))$ is the original connection of output signal with destination port of ρ , and M_δ is the destabilizer connection, $M_\delta = \{(\alpha, \sigma_o(\rho)), (\omega, s_\omega), (i_o(\rho), s_\omega)\} \cup f$, where:
 - α is connected to the original zero-delayed output of ρ ,
 - the new signal s_ω connects output ω of the destabilizer with the destination port of ρ ,

⁶Note that $[\rho]$ represents the set of all critical signal paths with the same destination port.

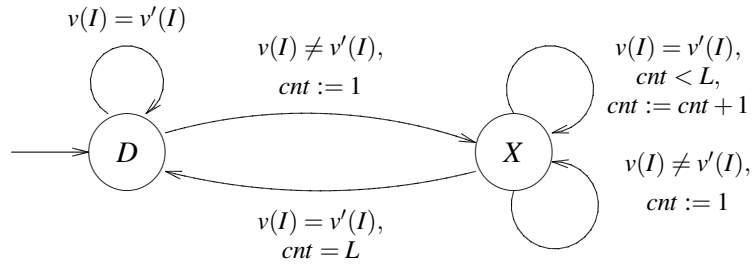


Figure 3.17: The automaton describing the behaviour of a destabilizer

- and f is a mapping of destabilizer’s monitored input ports to input signals of $[\rho]$ such that $f = \{(i_{\rho'}, \sigma_i(\rho')) \mid \rho' \in [\rho]\}$.

The behaviour of a destabilizer δ_ρ is defined by the finite automaton shown in Figure 3.17—for brevity, the automaton is described with one bounded counter whose possible values are not included directly in the state-transition control. Let $v(I) \neq v'(I)$ for a set of monitored inputs I denote that some input in I is changing its value (i.e., its current value differs from the value in the next step). If the destabilizer is in the D state—when it has a defined value (in particular, $\sigma_o(\rho)$) on its output—and one of the monitored signals changes its value, the destabilizer switches to the X state and produces on its output the x -value, i.e., randomly 0 or 1. The destabilizer will hold in the X state for a period of L steps where L is the number of critical input ports in the longest critical signal path of $[\rho]$ which the destabilizer is connected to, that is

$$L = \max(\{|i(\rho')| \mid \rho' \in [\rho]\}) \quad (3.2)$$

Implementation of a Destabilizer

Let us consider a general destabilizer for n critical signal paths with L being the length of the longest of these paths. In SMV, we can implement the destabilizer as the following module with input signals in_1, \dots, in_n to be connected to the monitored inputs of the covered critical paths, input port $alpha$ (the original output to be delayed), and the new output port $omega$. As the counter of steps of random values on the output ranges from 1 to L , we extend its range to 0 to L claiming that value 0 represents the D control state of the destabilizer. The code of a module is depicted in Figure 3.18.

A simple illustration of how a destabilizer is connected to the rest of a modelled design can be found in Appendix A.

```

1  module Destabilizer(in_1, ..., in_n, alpha, omega) {
2    input in_1 : boolean;
3    ...
4    input in_n : boolean;
5    input alpha : boolean;
6    output omega : boolean;
7    cnt : 0..L;
8    -- L is a constant value obtained from formula (3.2)
9
10   init(cnt) := 0;
11   next(cnt) := case {
12
13     -- one of the monitored signals is changing
14     ((in_1!=next(in_1)) |
15      ... |
16     (in_n!=next(in_n)))      : 1;      -- to state X
17
18     -- the counter reaches the maximum and
19     -- all monitored signals hold
20     cnt=L                    : 0;      -- to state D
21
22     -- the counter is in (0;L) and
23     -- all monitored signals hold
24     cnt>0 & cnt<L          : cnt+1; -- remain in X
25
26     -- all monitored signals are stable
27     1                       : 0;      -- remain in D
28   }
29
30   omega := case {
31     cnt=0 : alpha;
32     -- state D: propagate a defined value
33
34     cnt>0 : {0,1};
35     -- state X: output the x-value
36   }
37 }

```

Figure 3.18: The code of the module of a destabilizer in Cadence SMV. We believe that the function is clear from the definition of a destabilizer.

A Justification of the Construction

Modelling Capabilities. We are interested in checking that no dangerous stable combination of signals is reachable even though there is a possibility that some undefined signal values on critical signal paths will be sensed and registered. Therefore a method which over-approximates the influence of working with undefined signal values on the reachable stable combinations of signals is a sound solution.

A destabilizer is connected to the output of several critical paths. In the previously described method based on extending all critical input ports, it takes at most $L = |i(\rho)|$ steps to stabilize the output signal if the input signal of any critical path changes (provided ρ is the longest path of $[\rho]$). A destabilizer produces x-values for L steps if any of the input signals changes. Thus, the destabilizer will generate all the combinations of signals to be sensed and become stable as in the method based on extending all critical input ports and may be even more. Therefore, it is a safe over-approximation of the extension of all gates in critical signal paths.

However, if a model checker returns a counter-example in a model using destabilizers, we cannot be sure, due to the over-approximation, if it reflects a possible behaviour of the real system. In such a case, one must examine the counter-example manually or a more precise modelling method is required, e.g., we can use a model based on the extension of all critical gates and perform the verification once again. One could also think of performing the check only on the given path and possibly using the extension of all critical gates only on this path. A proper investigation of such an approach is a part of the future work.

Efficiency of the Method. We said that destabilizers often save a number of state variables compared to the method of extending all critical input ports. Let us examine when this approach is efficient wrt. the number of state variables. The method based on extending all critical input ports creates one new binary state variable per a critical input, i.e., $|\iota(H)|$ new state variables where $\iota(H)$ is the set of all critical input ports in a design H . The method using destabilizers also introduces new state variables. However, note that one destabilizer can replace the extension of all critical input ports that is needed in the first method on more than one critical signal paths. Let $L_{[\rho]}$, for $\rho \in \rho(H)$, represent the maximum number of input ports of critical signal paths $[\rho]$ as defined by formula 3.2. Every destabilizer for $[\rho]$ adds to the system a counter of unstable values of the range $[0; L_{[\rho]}]$, i.e., the size of the counter is $\lceil \log_2(L_{[\rho]} + 1) \rceil$ bits. In order to get the number of new state variables for the whole hardware design H , we need to summarize newly added variables for all the sets of critical signal paths. For such a purpose, we define $R = \{[\rho] \mid \rho \in \rho(H)\}$ being the sets of critical signal paths sharing the same destination port. Then, the number of state variables added to the design by the destabilizer method is $\sum_{[\rho] \in R} \lceil \log_2(L_{[\rho]} + 1) \rceil$. Finally, we can say that the method of extending critical signal paths is more efficient than the method of extending all critical input ports if

$$\sum_{[p] \in R} \lceil \log_2(L_{[p]} + 1) \rceil < |v(H)| \quad (3.3)$$

This typically fits for a design with long critical signal paths (delayed inputs connected in cascade are reduced) or with a lot of critical signal paths leading to the same input port (one destabilizer substitutes a number of delayed inputs). Our experiments (described in detail in Section 3.5) show that, when dealing with real-life component of asynchronous FIFO, the method of destabilizers gains this advantage.

3.3.5 Modelling with One-step Destabilizers

In the third method of modelling data transfer between two clock domain, we focus on the number of random outputs generated by the destabilizer described above. Our goal is to show that only one step of generating random values is sufficient for most hardware designs. In this section, we present a method of modelling using a destabilizer providing a random value for a single step only. We will justify our construction by showing that one step of random values on the output of critical signal path is sufficient to reveal inconsistencies caused by design faults in clock domain crossing.

Clock Constraint

When modelling the environment as described in Section 2.2, it is possible that clock signals are being random all the time. In such a case, the interval between two rising edges of a clock signal may be two verification steps at least (at first step, the clock signal changes its value from low to high; in the next step, the value falls down). Both methods of handling asynchronicity that we proposed delay the signal value at the destination of a critical signal path before its propagation to another clock domain for L verification steps at most where L is the length of the critical signal path. If $L > 2$, it is possible for the destination clock domain, to read the inputs from the signal path during its stabilization phase more than once (in particular, n -times where $1 \leq n \leq \lceil L/2 \rceil$).

For most hardware designs, it is uncommon that the delay of a propagation of a stable value through a signal path is greater than a short interval of two clock ticks. In such a case, proposed methods may introduce false alarms as they unnecessarily over-approximate the model. Moreover, such an over-approximation may complicate the reachability analysis by a need to explore runs infeasible in the real system.

It is possible to remove infeasible behaviour by providing only a single step of randomizing the value of critical signal path such that the signal leading to a destination clock domain is stabilized within two verification steps. We now concentrate on optimizing our methods for those hardware designs in which every critical signal path propagates a new signal within a delay less than a clock period of a destination

clock domain. Knowing the frequency and electric parameters of the technology that is used, it is possible to determine the validity of such a condition [42, 43, 44].

A One-step Destabilizer

In the following, we describe an modification of the model with destabilizers that produces random outputs for only one verification step—we call them *one-step destabilizers*. The extension of a zero-delayed model is performed in a similar way as described in the previous section dealing with extension of critical paths.

As before, a one-step destabilizer added to the output of a critical signal path $\rho \in \rho(H)$ randomizes the output shared with critical signal paths $[\rho]$ (the behaviour of a one-step destabilizer will be described later). We define a one-step destabilizer as a gate:

$$\delta_1 = (\{\perp\}, \{i_1, \dots, i_n, \alpha\}, \{\omega\})$$

where $I' = \{i_1, \dots, i_n\}$, $I' \cap P = \emptyset$, is the set of new ports of monitored inputs, α is a new port sensing the zero-delayed output of ρ , and ω is a new port which delays the input for a single step.

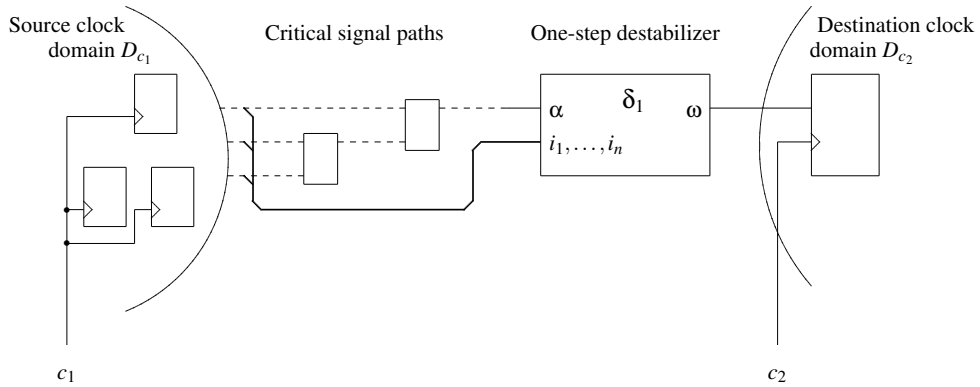


Figure 3.19: A schematic connection of a one-step destabilizer to critical signal path(s).

A symbolic scheme of one-step destabilizer is depicted in Figure 3.19. Like in the case of using plain destabilizer, we add a one-step destabilizer to every set of critical signal paths sharing the same destination port. The modification of a hardware design is the same, we introduce it only for a complete picture: For a set of critical signal paths with the same destination port $[\rho]$, we change the hardware design $H = (S, C, P, G, M)$ to $H' = (S', C, P', G', M')$ where:

- $S' = S \cup \{s_\omega\}$, $s_\omega \notin S$, is the set of signals containing new signal connecting the one-step destabilizer with the destination port of ρ ,

- $P' = P \cup I_\rho \cup \{\alpha, \omega\}$, $I_\rho = \{i_{\rho'} \mid \rho' \in [\rho]\}$, $(I_\rho \cup \{\alpha, \omega\}) \cap P = \emptyset$, is the set of ports with the new ports of δ_1 ,
- $G' = G \cup \{\delta_1\}$ is the set of gates with the one-step destabilizer $\delta_1 = (\perp, I_\delta \cup \{\alpha\}, \{\omega\}) \notin G$, and
- $M' = (M \setminus \{(i_o(\rho), \sigma_o(\rho))\}) \cup M_\delta$ is the port–signal mapping where $(i_o(\rho), \sigma_o(\rho))$ is the original connection of output signal with the destination port of ρ , and M_δ is the one-step destabilizer connection, $M_\delta = \{(\alpha, \sigma_o(\rho)), (\omega, s_\omega), (i_o(\rho), s_\omega)\} \cup f$, where:
 - α is connected to the original zero-delayed output of ρ ,
 - the new signal s_ω connects output ω of the one-step destabilizer with the destination port of ρ ,
 - and f is a mapping of one-step destabilizer’s monitored input ports to input signals of $[\rho]$ such that $f = \{(i_{\rho'}, \sigma_i(\rho')) \mid \rho' \in [\rho]\}$.

The behaviour of a one-step destabilizer is defined by the automaton depicted in Figure 3.20. The states of the automaton represent the next value of output ω . Let $I = \{i_1, \dots, i_2\}$, the expression $v(I) = v'(I)$ then stands for a constraint that there is no change of values of any of the monitored input ports. On the other hand, $v(I) \neq v'(I)$ means that one or more monitored signals connected to the input ports I change. In a stable state, a one-step destabilizer produces the α value—the input of a one-step destabilizer and a value of the output signal of a critical signal path. If at least one of the monitored inputs changes, the one-step destabilizer produces a random output and the destination clock domain is able to receive either 0 or 1. If this change is caused by a single clock domain, the one-step destabilizer produces a random phase *for a single verification step*—a change of signals of a source clock domain is caused by rising (or falling) edge; the rising (or the falling respectively) edge cannot happen immediately in the next step, thus the monitored signals hold their value and the one-step destabilizer switches back to a stable state.

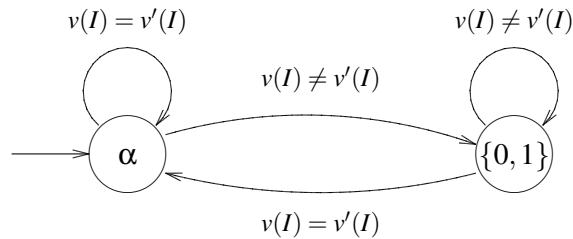


Figure 3.20: The automaton controlling the output of a one-step destabilizer.

```

1 module OnestepDestabilizer(i_1, ..., i_n, clk, alpha, omega) {
2   input i_1 : boolean;
3   ...
4   input i_n : boolean;
5   input clk : boolean;
6   input alpha : boolean;
7   output omega : boolean;
8
9   init(omega) := alpha;
10
11  next(omega) := case {
12
13    -- one of the monitored signals is changing
14    ((i_1 != next(i_1) |
15     ...
16     (i_n != next(i_n))      : {0,1};
17
18    -- otherwise output the stable input
19    1                          : alpha;
20  }
21 }

```

Figure 3.21: The code of the module of a one-step destabilizer.

Implementation of One-step Destabilizers

We believe that the implementation of one-step destabilizer in Figure 3.21 is straightforward for the reader. Compared to a plain destabilizer, it introduces a new one-bit state variable representing output ω of the finite automaton in Figure 3.20.

An example of how to include one-step destabilizer into a design would be very similar to the example of using destabilizer, which is shown in Appendix A, and here we do not repeat the exercise.

A Justification of the Construction

In the next paragraphs, we provide a justification of the method of extending model with one-step destabilizers. We aim at modelling capabilities and at a case when the method is more efficient wrt. previously proposed method of plain destabilizers. Just for a complete picture of the construction, we discuss the differences between one-step destabilizers and delayed inputs.

Modelling Capabilities. The extension using one-step destabilizers is justified by our assumption that a propagation delay of any critical signal path of a circuit is less

than the clock period of a destination clock domain. This means that we assume that unstable values caused by a change due to a single event of a source clock domain (rising or falling edge of a clock) can be sensed by a destination clock domain once only for that event (such an assumption can be checked for a specific hardware design by calculating a propagation delay using the specification of the technology).

The proposed modification of a model makes the destination port of a critical signal path random *if a change is sensed at the source of a critical signal path*. In a circuit, if a change is generated at the source of a critical signal path, an unstable value caused by propagating this change through a path may be sensed by a destination clock domain. In such a case, a one-step destabilizer will provide the x-value at a destination port of a critical signal path, which simulates reading of an unstable value by a destination clock domain.

A random phase of one-step destabilizer is produced *as long as the inputs change* (cf. Figure 3.20). The length of a random phase needs some further consideration. If a design includes an intersection of critical signal paths described on page 39, we must ensure that a change caused by a source clock domain cannot generate unstable value readable also by the same clock domain in its next clock event. In an extended model, if a change is caused by *a single critical signal path*, the responsible one-step destabilizer produces a random value *for a single verification step* since the event that causes such changes cannot happen again immediately in the next step, and the one-step destabilizer stabilizes its output because no other change is sensed. Thus, a destination port will be already stabilized in the next clock event. On the other hand, a random phase of a one-step destabilizer may *last more verification steps*. This can happen only if it monitors signals from more source clock domains and one clock domain changes its signals one step after another clock domain did so. In such a case, a longer random phase is also justified as a one-step destabilizer produces an unstable value leading to a clock domain that *did not invoke the change*.

We have shown that a model extended with one-step destabilizers is a safe over-approximation. This over-approximation is also the main disadvantage of the method because it may produce false alarms. Thus, one must examine every counter-example which can be done in a similar way discussed in the method of extending critical signal paths with plain destabilizers.

Efficiency of the Method. Each instance of a one-step destabilizer creates a new one-bit state variable. The number of instances of a one-step destabilizer, i.e., the number of introduced state variables in the hardware design H is $|R|^7$. Since the method of extending critical signal path (with plain destabilizers) has very similar modelling capabilities (plain destabilizers just produce more random scenarios as the duration of a random phase is longer), we discuss the case when the proposed method is more

⁷Remember that $R = \{[\rho] \mid \rho \in \rho(H)\}$ includes all sets of critical signal paths sharing the same destination port.

efficient in a number of introduced variables. From the formula 3.3, we can state that the method of one-step destabilizer is more efficient if:

$$\sum_{[\rho] \in R} \lceil \log_2(L_{[\rho]} + 1) \rceil > |R|$$

From the definition of a critical signal path (here, note that every critical signal has at least one critical input port), we modify the condition into the form:

$$\exists r \in R : \lceil \log_2(L_r + 1) \rceil > 1$$

that is $\exists r \in R : L_r > 1$, which can be even simplified as:

$$\exists \rho \in \rho(H) : |i(\rho)| > 1 \quad (3.4)$$

This means that the method of one-step destabilizer is more efficient on a hardware design which contains at least one critical signal path with two or more input ports. Since every critical signal path contains one input port at its destination, more inputs are the part of a combinational logic. That is, the new method is more efficient if a critical signal path *contains one or more combinational gates*.

Delayed Input (Δ) versus One-step Destabilizer (δ_1). Since both modules of a delayed input gate and a one-step destabilizer are very similar, one can wonder why the delayed input gate cannot be used at the end of a critical signal path instead of a one-step destabilizer. Both gates delay its input by a precisely one step during which a random output is produced. The main difference is *when* the delay (or a random phase) is generated. A one-step destabilizer produces a random output whenever any of sources of the critical signal paths it represents changes, but a delayed input gate is restricted only to the change of the gate on its input. Next, we show a simple example where there is a difference between the behaviour of the design extended by a single delayed input gate and a one-step destabilizer.

Consider two critical signal paths starting with input signals a and b , and sharing the same destination port connected to a signal d . The function of the logic circuit is described in an RTL design by logic equations $c = \neg b$, $d = a \wedge c$, $x = d$, and is schematically depicted in Figure 3.22.

A real logic circuit produces a short glitch on signal d if the value of signals a and b changes from 0 to 1 at the same time. Just for simplicity of an explanation, we consider that the propagation delays of all gates are equal. The hardware state of the circuit in time t is then described by equations:

$$c_t = \neg b_{t-1}, \quad d_t = a_{t-1} \wedge c_{t-1}, \quad x_t = d_t$$

Next, consider a scenario with the initial state $a_0 = 0$, $b_0 = 0$, $c_0 = 1$, $d_0 = 0$, and $a_t = b_t = 1$ for $t > 0$. The signal d will hold value 1 for a while due to the propagation delay of the NOT gate:

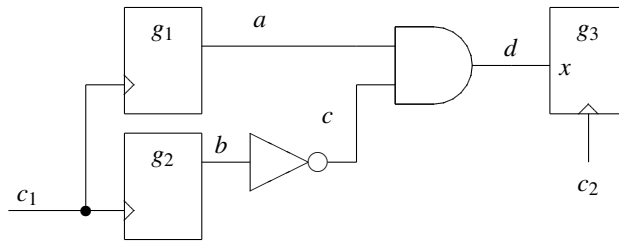


Figure 3.22: A simple example of two signal paths of the clock domain crossing.

$t = 0 :$	$a_0 = 0$	$b_0 = 0$	$c_0 = 1$	$d_0 = 0$	$x_0 = 0$	stable
$t = 1 :$	$a_1 = 1$	$b_1 = 1$	$c_1 = 1$	$d_1 = 0$	$x_1 = 0$	unstable
$t = 2 :$	$a_2 = 1$	$b_2 = 1$	$c_2 = 0$	$d_2 = 1$	$x_2 = 1$	unstable
$t = 3 :$	$a_3 = 1$	$b_3 = 1$	$c_3 = 0$	$d_3 = 0$	$x_3 = 0$	stable

The zero-delay model of such a circuit changes the logic a bit:

$$c_t = \neg b_t, \quad d_t = a_t \wedge c_t, \quad x_t = d_t$$

so signal d holds value 0 all the time. A *single* delayed input connected between signal d and input port x senses no change on d , thus produces a non-randomized sequence of zeros:

$t = 0 :$	$a_0 = 0$	$b_0 = 0$	$c_0 = 1$	$d_0 = 0$	$x_0 = 0$	stable
$t = 1 :$	$a_1 = 1$	$b_1 = 1$	$c_1 = 0$	$d_1 = 0$	$x_1 = 0$	stable

(just for a complete picture, if the delayed-input gates were connected also to signals b and c , a random value would be produced).

On the other hand, a one-step destabilizer is sensitive on inputs of the critical signal paths it represents, i.e., to signals a and b too. Thus, the one-step destabilizer will output a random value at time $t = 1$ (at time $t = 0$, the one-step destabilizer senses a change of signal a , so the next value of its output x_1 will be random). Then, the output will be immediately in the next state stabilized at value 0 (for time $t > 0$, the one-step destabilizer senses no change on monitored input and thus provides stable output $x_{t+1} = 0$):

$t = 0 :$	$a_0 = 0$	$b_0 = 0$	$c_0 = 1$	$d_0 = 0$	$x_0 = 0$	stable
$t = 1 :$	$a_1 = 1$	$b_1 = 1$	$c_1 = 0$	$d_1 = 0$	$x_1 \in \{0, 1\}$	unstable
$t = 2 :$	$a_2 = 1$	$b_2 = 1$	$c_2 = 0$	$d_2 = 0$	$x_2 = 0$	stable

The non-determinism at time $t = 1$ provides two possible traces, one of which reveals a glitch on input port x . Both traces do not equal to the trace of a hardware state of the circuit, but as we are interested only in possible values of input x of the destination clock domain, the traces conform to scenario possible in the circuit.

3.3.6 Delaying the Clock Domain Output

When comparing the use of one-step destabilizers with the other methods, its main advantage is that it dramatically reduces the number of new state variables—just one state variable per the output of a group of critical signal paths is introduced. The disadvantage is that it introduces false alarms. These false alarms make troubles especially when verifying a synchronization implemented by a combinational logic changing one signal value at a time. More precisely, the problem appears in critical signal paths with the same destination port holding its value even if some source ports of the signal paths sense a value change⁸. In such a case, a one-step destabilizer senses the change on the monitored inputs and produces a (false) signal change at the destination port.

In this section, we provide our fourth method of describing clock domain crossing circuits. The method is similar to using one-step destabilizers in the number of the introduced state variables, but has more precise modelling capabilities. The main idea is based on one-step destabilizers but differs in the critical parts of the design the method extends. In the method of one-step destabilizers, only critical input ports at the end of critical signal paths are extended. A one-step destabilizer then generates random values on the input of the destination clock domain each time the source signals change, i.e., a one-step destabilizer assumes that a change of a signal originating from a different source clock domain could lead to a change of the destination port. However, randomizing the destination whenever the source changes is a very coarse over-approximation of the system behaviour. In particular, such an over-approximation complicates verification of circuits consisting combinational logic designed such that they “consume” the change of a signal, i.e., logic gates that do not propagate the change to the destination clock domain.

To refine such a model, we use the logic of the combinational gates to restrict the randomness. A natural way to do so is to use zero-delayed model of the combinational logic at critical signal paths and to randomize its inputs only (in contrast to extending the zero-delay behaviour of the outputs at the method of one-step destabilizers). More precisely, we let every source port of every critical signal path random every time it changes its value and we let this random value be sensed by the rest of the combinational logic of the critical signal path. If the combinational logic is designed such that it does not propagate a change further to the system, the destination port of the critical signal path does not sense the change.

To produce random values on the output of clock domains, we can use the *delayed input gate* (see Section 3.3.3). Conceptually, the delayed input differs from a one-

⁸A typical example of such a logic is the transfer of a binary counter value from one clock domain to another using the Gray code—a binary code of an integer number in which the Hamming distance of two adjacent numbers equals to one. The transfer of an incremented or decremented counter value causes a change on one signal only, and so this, from the definition of clock domain crossing, does not imply synchronization problems.

step destabilizer in the set of the monitored inputs. In case of delaying a signal just at the source of a critical signal path, a delayed input and a one-step destabilizer represent the same behaviour. In the following, we will call the delayed input gate as a *delaying gate* since it can be simply misunderstood as it is connected to the *output* of a clock domain.

The schematic idea of the method is depicted in Figure 3.23 (compare it with the schematic use of a one-step destabilizer in Figure 3.19). As we extend the model with delaying gates at places which transmit data *out* from a clock domain, we call the method as *delaying the output of a clock domain*.

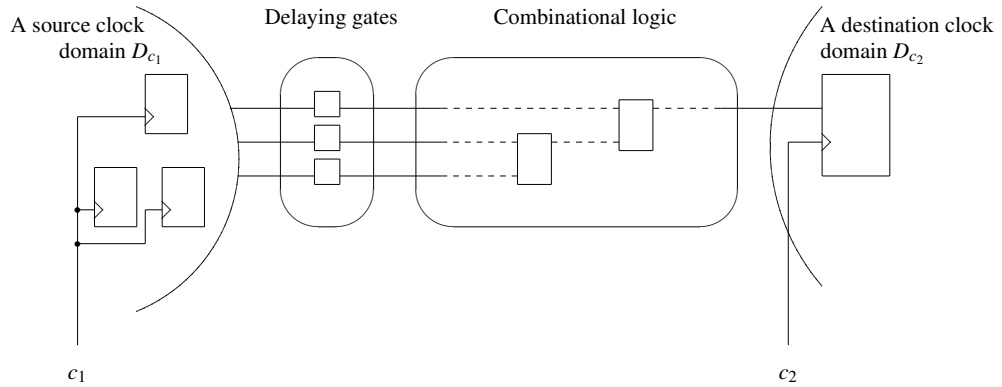


Figure 3.23: The idea of delaying clock domain output with delaying gates. Delaying gates are connected the source ports of critical signal paths, i.e., to the output of a source clock domain.

Formally, for every critical signal path $\rho \in \rho(H)$ of a hardware design $H = (S, C, P, G, M)$, we transform H to $H' = (S', C, P', G', M')$ by adding a delaying gate Δ as follows:

- $S' = S \cup \{s'\}, s' \notin S$, is the set of signals containing a new signal connecting the delaying gate with the extended input port.
- $P' = P \cup \{i', o'\}, i', o' \notin P$, is the set of ports extended by input i' and output o' port of the delaying gate that we are adding.
- $G' = G \cup \{\Delta\}$, is the set of gates with the new delaying input $\Delta = (\perp, \{i'\}, \{o'\}) \notin G$.
- $M' = (M \setminus \{(i_2, s_1)\}) \cup M_\Delta$ is the new mapping of ports to signals where $i_2 = i_i(\rho)$ is the source port of the path ρ and s_1 is the signal connected to it, i.e., $\rho = \langle g_1 o_1 s_1 i_2 g_2 \dots i_n g_n \rangle$, and $M_\Delta = \{(i', s_1), (o', s'), (i_2, s')\}$ is the interconnection of the ports and signals around the delaying gate (cf. Figure 3.24).

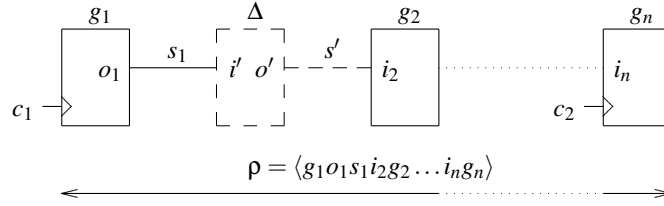


Figure 3.24: A schematic description of the interconnection of a delaying gate to a critical signal path. The dashed part is being added to the design and represents the delaying gate including the new signal.

A Justification of the Construction.

In the following, we will justify our construction on the modelling capabilities. We will not discuss the length of a random phase (the delay of a propagation of a signal) since it has the same effect as in the method using one-step destabilizers.

Modelling capabilities. The proposed method extends the zero-delay model of a hardware design in which the transitional effect during which a signal can change its value is lost. Since we let most of the logic of a critical signal path zero-delayed, we build our justification on the assumption that the examined hardware design is hazard free (i.e., there is no static or dynamic hazard) [59]. That means that delaying clock domain output does not necessarily produce random values which are caused by transitional glitches in a combinational logic.

Hazard free critical signal paths with the same destination port produce a change on their output if and only if a change is sensed on one or more of their inputs. If *only one critical signal path* changes its input, a delaying gate connected to its source port produces a random value for a single verification step. Such a delay then makes the transitional effect of a change visible for the destination clock domain.

An interesting situation is when *two critical signal paths* ρ_1 and ρ_2 sharing the same destination port, $[\rho_1] = [\rho_2]$, change their inputs at the same time t (a generalization to more paths is straightforward). In such a case, a combinational logic of both paths may produce an output which corresponds to four different configurations on their inputs:

1. $(i_i^t(\rho_1), i_i^t(\rho_2))$,
2. $(i_i^{t+1}(\rho_1), i_i^t(\rho_2))$,
3. $(i_i^t(\rho_1), i_i^{t+1}(\rho_2))$, and
4. $(i_i^{t+1}(\rho_1), i_i^{t+1}(\rho_2))$

where $i_t^t(\pi)$ represents a value of the source port of the path π at time t and $i_t^{t+1}(\pi)$ represents a value in the next step. Both delaying gates connected to sources of paths ρ_1 and ρ_2 will produce random values on their outputs, thus all four configurations are reachable. Therefore, it is a safe over-approximation which is similar to the method using one-step destabilizers.

On the other hand, an over-approximation of the method of delaying clock domain output is not so coarse as in the previously proposed method. The method of delaying the output of a clock domain models in a more refined way a design containing a critical signal path which *consumes a change on its input*, i.e., no change is sensed on its output. Since the combinational logic of a path is modelled using the zero-delay abstraction, a random value produced by a delaying gate is *hidden* by the combinational logic. Comparing to the previous method, a one-step destabilizer *always* produces a random value on the destination port of a critical signal path.

3.4 Algorithms of Finding Critical Ports

Every proposed modelling method has been described as an extension of a critical signal path or a part of it. The implementations of all the methods require finding all critical signal paths of a design and finding their critical input ports. In the next section, we provide an algorithm for finding critical input ports. Then, we describe how we can filter the set of critical input port to be used in every modelling method (e.g., to search for critical input ports at the source or at the destination of critical signal paths) with no need to find critical signal paths.

Before specifying the set of critical input ports, we introduce some definitions which give us the better notion of the structure of a hardware design $H = (S, C, P, G, M)$:

- Let $P_I = \bigcup_{(c,I,O) \in G} I$ denote the set of all input ports.
- Let $\mathcal{S} : 2^P \rightarrow 2^S$ represent a mapping which signals are connected to a subset of ports, i.e., $\mathcal{S}(Ports) = \bigcup_{p \in Ports} \{M(p)\}$.
- Further we define $Pred : S \rightarrow 2^S$ as a function that returns for a signal s the set of signals which are connected to the inputs of a gate that sets signal s , i.e., for $s \in S$: $Pred(s) = \{s' \in S(I) \mid \exists (c, I, O) \in G : s \in S(O)\}$.
- We also define $Succ : P \rightarrow 2^P$ as a function of successors of an input port. Successors of input port p of gate g are input ports to which g is connected with its outputs, i.e., for $p \in P_I$: $Succ(p) = \{p' \in P_I \mid \exists (c, I, O) \in G : p \in I \wedge M(p') \in S(O)\}$. An example of predecessors and successors are depicted in Figure 3.25.

Next, we calculate a mapping $SC : S \rightarrow 2^{C^\perp}$ which describes, for a signal s , which clock signals influence the next value of s . We also define a mapping $SC_0 \subseteq SC$ which

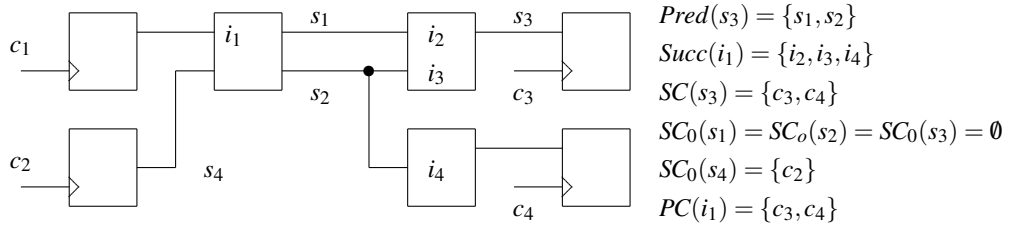


Figure 3.25: An example of predecessors of a signal, successors of an input port, and an influence of signals, clock signals, and input ports.

represents, by a clock signal reference, for a signal s which clock domain includes a gate defining a value of the signal s , i.e., for $g = (c, I, O) \in G$ and $s \in \mathcal{S}(O)$: $SC_0(s) = \{c\} \setminus \{\perp\}$. The calculation of SC is a bit complicated and is described by Algorithm 3.1.

Similarly to SC , we calculate a mapping $PC : P_I \rightarrow 2^{C_\perp}$ which represents, by a set of clock signals, for an input port p , which clock domains can sense a value change of p . The calculation is similar to the calculation of SC and is described by Algorithm 3.2. For an illustration of mappings SC , SC_0 , and PC see Figure 3.25.

Algorithm 3.1

Input: A set of signals S and mappings S , SC_0 , and $Pred$.

Output: The mapping SC .

```

1  CombOutput := {  $s \in S \mid SC_0(s) = \emptyset$  }
2   $SC := \emptyset$ 
3   $SC' := SC_0$ 
4  while  $SC \neq SC'$  do
5     $SC := SC'$ 
6    for all  $s \in CombOutput$  do
7      for all  $s' \in Pred(s)$  do
8         $SC'(s) := SC'(s) \cup SC'(s')$ 
9      end for
10   end for
11 end while

```

Description: When searching for clock signals that may influence a value of a signal, we go *backward* to the propagation of a signal using the mapping $Pred$. Since every signal that directly outputs from a synchronous gate preserves its mapping SC from the initial one SC_0 (only one gate may set a value of a signal and if the gate is synchronous, the signal is triggered by a clock of that gate only), it is required to calculate the mapping SC for signals set by combinational gates only. For that purpose, we create a set of signals *CombOutput* (line 1) over which we calculate the mapping

SC . The calculation is performed by finding a transitive closure of SC in an iterative way (lines 4–11) starting from the initial mapping SC_0 (lines 2–3) such that there is no combinational gate whose inputs and outputs are sensitive on different sets of clock signals (lines 7–9).

Algorithm 3.2

Input: A set of gates G , a set of input ports P_I , and a mapping $Succ$.

Output: The mapping PC .

```

1 for all  $g = (c, I, O) \in G$  do
2   for all  $p \in I$  do
3      $PC_0(p) := \{c\} \setminus \{\perp\}$ 
4
5    $CombPorts := \{ p \in P_I \mid PC_0(p) = \emptyset \}$ 
6    $PC := \emptyset$ 
7    $PC' := PC_0$ 
8   while  $PC \neq PC'$  do
9      $PC := PC'$ 
10    for all  $p \in CombPorts$  do
11      for all  $p' \in Succ(s)$  do
12         $PC'(s) := PC'(s) \cup PC'(s')$ 
13      end for
14    end for
15  end while

```

Description: The algorithm is very similar to Algorithm 3.1. Instead of using the mapping SC_0 , we calculate an initial mapping PC_0 (lines 1–3) which represents, for an input port p of a synchronous gate, which clock signal triggers reading of a value of p . We then calculate a transitive closure of the mapping PC (lines 6–15) by *forward* searching for the destination clock domain using the mapping $Succ$. We calculate the mapping over a set of input ports of combinational gates only (lines 7 and 10) since $PC(p)$ for the port p of a synchronous gate equals to $PC_0(p)$.

Finally, we generate the set of all critical input ports. We can define a critical input port using mappings PC and SC as follows: If more clock domains are sensitive on a value of an input port $p \in P_I$, i.e., $|PC(p)| > 1$, the port p is clearly critical. If just one clock domain is sensitive on a value of the port p and the signal connected to the port p is not influenced by this clock domain only, i.e., $|PC(p)| = 1 \wedge PC(p) \neq SC(M(p))$, the port is critical since it lays on a critical signal path leading from the clock domain of $PC(p)$ and to a clock domain of $SC(M(p))$ different from $PC(p)$. Thus, we construct a set of all critical input ports

$$CriticalPorts = \{p \in P_I \mid |PC(p)| > 1 \vee (|PC(p)| = 1 \wedge PC(p) \neq SC(M(p)))\}$$

Each proposed modelling method selects a different subset of the critical input ports computed as described above:

- *Extending all critical input ports* simply uses every input port from the set *CriticalPorts* and creates an instance of a delayed input connected to the port.
- *Extending critical paths* uses the set of critical *destination* ports. A destination port is a critical input port that belongs to a synchronous gate:

$$CriticalDestinationPorts = \{p \in CriticalPorts \mid \exists(c, I, O) \in G : p \in I, c \neq \perp\},$$
i.e., all critical input ports of a gate whose clock domain is different from \perp . The method also needs to know the length of the longest critical signal path the destabilizer represents. The number of gates in the longest critical signal path can be simply calculated by backtracking on critical signal paths from the destination port to different clock domains using the *Pred* function and the mapping SC_0 identifying which signals are directly set by a gate of a clock domain different from \perp .
- *Modelling with one-step destabilizers*, similarly to the method of extending critical paths, extends the destination critical ports and so the same set of critical input ports is used—*CriticalDestinationPorts*. Moreover, each one-step destabilizer requires the set of monitored inputs to be connected to. A simple way to obtain such a set is to compute it in a similar way to calculating the length of the critical signal path, i.e., by backtracking on all critical signal paths with the same destination port, with the exception that we do not calculate the number of critical input ports on signal paths, but search for last predecessors (i.e., search for source ports of critical signal paths).
- *Delaying clock domain output* connects a delaying gate to the source port of each critical signal path. A source port is a critical input port which is connected to a signal set by a synchronous gate, i.e., it is connected to a signal whose initial mapping SC_0 is set to any clock signal: $CriticalSourcePorts = \{p \in CriticalPorts \mid SC_0(M(p)) \neq \emptyset\}$.

3.5 A Comparison of Modelling Methods

In this section, the modelling capabilities of the proposed methods are shown on experiments with models of two different synchronization mechanisms. Further, a recommended approach to functional verification of asynchronous components is discussed.

As most hardware designs are component-based, it is common that the synchronization circuit between two or more clock domains is managed by a single synchronization component. The experiments of the proposed methods were performed on two such components—(i) a synchronization module using a simple handshake protocol, and (ii) an asynchronous FIFO unit.

We implemented all proposed modelling methods in the tool *CDCreveal* (available online in [90]) which uses Cadence SMV as an input language. We successfully

used the tool on both asynchronous units. The next tables show the statistics of a verified model and the size of its state space:

- The first column identifies the used *method*:
 - the row *no extension* represents the statistics of the original zero-based model,
 - *all critical ports* stands for the model produced by the method of extending all critical input ports,
 - *destabilizers* represents the method of extending critical signal paths with destabilizers,
 - *one-step destabilizers* for modelling critical signal paths using one-step destabilizers, and
 - *clock domain outputs* for the statistics of model where clock domain outputs were extended.
- The column *variables* shows the number of binary state variables of the model.
- The *trans. relation* is the number of nodes of the BDD encoded transition relation.
- The column *reach. set* represents the number of BDD nodes of the set of reachable states.
- *Satisfied* shows if the verified property was satisfied in a model.

The time spent by model checking of our two examples was below 2 seconds for all cases (except one special case of asynchronous FIFO discussed below), thus this data does not carry any useful information.

Details of the simple handshake protocol are described in Appendix B. Briefly described, it is a component which transmits two binary signals from one clock domain to another. For a handshake, the component uses the control signal `rdy` (data is ready) coming from a source clock domain and the control signal `ack` (data is received) coming from a destination clock domain. The component is interesting in a length of critical signal paths. There are four critical signal paths (a buggy protocol includes three critical signal paths), each of them has one critical input port only (i.e., there is no combinational logic in a clock domain crossing).

The property being verified aims at consistency of a transfer of two-bit data (for more details see Appendix B). The results for the good version of a handshake protocol show that the property is satisfied in all experiments. However, when dealing with buggy handshake protocol, zero-delayed model hides the CDC problem. Here, when we apply any of the proposed method, the fault will become visible. All of

method	variables	trans. relation	reach. set	satisfied
no extension	18	194	2251	true
all critical ports	22	226	4255	true
destabilizers	28	266	8431	true
one-step destabilizers	22	226	4255	true
clock domain outputs	22	226	4255	true

(a) A correctly implemented handshake

method	variables	trans. relation	reach. set	satisfied
no extension	17	180	1598	true
all critical ports	20	204	4016	false
destabilizers	25	240	7817	false
one-step destabilizers	21	212	4538	false
clock domain outputs	20	204	4016	false

(b) A buggy handshake protocol

Table 3.2: Experimental results with model checking a simple handshake synchronization protocol.

the proposed methods produce the same or very similar models wrt. number of introduced state variable or the size of transition relation (cf. Table 3.2) since the design contains very simple clock domain crossings.

The asynchronous FIFO is a component from the Liberouter project [80] in which it is commonly used as a synchronization unit in several designs. The FIFO instantiates a dual-port memory for sending and receiving information items, the control part of FIFO uses read and write counters (for addressing the memory) and signals `full` and `empty` informing about the state of FIFO. Values of these signals depend on the result of a comparison of the read and the write counters. Due to different clocks controlling the value of counters (read clock controls the reading counter, write clock controls the writing counter), their comparison must be performed in a synchronized clock domain (signal `full` is set in the writing clock domain, signal `empty` in the reading clock domain). The Gray code is typically used for this purpose. The correct version of the FIFO circuit implements the Gray code in a proper way, the other version does not Gray-code a transferred counter value. The only clock domain crossing is in the part of the design which computes the signals `full` and `empty`.

The property being verified specifies the proper settings of signal `full`, in particular, that the FIFO does not notify that it is full if there are at least two non-allocated items in the FIFO's memory. The property itself is quite complex—we refer the reader to the set of tests of the tool *CDCreveal* for details (the tool and the

method	variables	trans. relation	reach. set	satisfied
no extension	41	873	32224	true
all critical ports	179	3123	N/A	N/A
destabilizers	61	1269	164042	false
one-step destabilizers	43	953	37314	false
clock domain outputs	73	1691	65533	true

(a) A correctly implemented synchronization

method	variables	trans. relation	reach. set	satisfied
no extension	41	845	33213	true
all critical ports	179	3111	N/A	N/A
destabilizers	61	1241	155346	false
one-step destabilizers	43	925	69823	false
clock domain outputs	73	1659	70134	false

(b) A wrong synchronization of counters

Table 3.3: Experiment results of model checking the asynchronous FIFO.

experiments are available online in [90]).

Considering the results of the property satisfaction in Table 3.3, the zero-delayed model of the good and the buggy version of the FIFO shows that the property is satisfied. The “not available” (N/A) information in case of *extending the model with delayed inputs* at every critical input port means that the model was so complex that the Cadence SMV model checker exceeds 4GB memory limit while computing the set of reachable states. In comparison with the previous synchronization protocol, here, *modelling using destabilizers* is more efficient than extending all critical input ports (because of the length of the signal path of the logic transforming the binary code of counters to the Gray code). The interesting results provides the *method of one-step destabilizers*. For both versions of FIFO, CDCreveal found only two sets of critical signal paths with the same destination port, thus only two state variables were introduced. Unfortunately, both methods using plain or one-step destabilizers produce false alarms over the correct version of FIFO. On the other hand, also interesting is the method of *delaying clock domain output*. Even though the number of state variables is almost twice the size the variable number of a model with one-step destabilizers, the overall complexity of the model is held down. Moreover, the property is satisfied on the correct version of FIFO but the CDC problem is revealed when faulty synchronization is used.

3.5.1 A Methodology of Formal Verification of Asynchronous Systems

Most of the components developed in RTL work in the same clock domain. The communication of two components working in different clock domains is in general solved either (i) via special asynchronous data transfer component or (ii) using some kind of data coding or synchronization protocol. In the following, we will propose the approach to functional verification of both such solutions.

Verifying Asynchronous Components

Formal verification of the first type of clock domain crossing can be performed in two steps: (1) verify that the asynchronous component itself works properly (using some of the model extensions to make CDC problem visible in a model), and (2) verify that the communication of the asynchronous component with the rest of the design meets its specification. According to the presented experimental results, it is reasonable to use either one-step destabilizer approach, or to delay clock domain output. The user must remember that one-step destabilizer method is sound but may produce false alarms. On the other hand, delaying clock domain output does not provide false alarms, but it can fail to notice a logic hazard. Therefore, one must take special care to check whether the logic is hazard-free or not. Moreover, if the propagation delay of some critical signal path is greater than the length of a clock period of a destination clock domain, neither one-step destabilizers or delaying gates can be applied. The only methods that reveals CDC problems properly are extending using plain destabilizers and extending all critical input ports. Extending using destabilizers is more efficient but over-approximates the original model a lot. Extending all critical input ports is quite precise modelling method but it may cause the state-space explosion.

Verifying Asynchronous Communication

Considering a user-defined synchronization protocol (i.e., two synchronous components working in different clock domains communicate with each other via some kind of synchronization protocol), there is no universal approach to verify the synchronization since the communication between two clock domains is user-specific. In such a case, one must (1) focus on signals of a protocol (i.e., signals on critical signal paths), (2) specify a property of a consistent data, and (3) specify an environment. All these steps must be performed manually.

In case of verifying that a component implements properly the transmitting or receiving part of a synchronization protocol, most likely the most problematic part of the verification will be the specification of an environment. To properly verify the communication, one must *exemplary* model the complement of a synchronization protocol.

If critical signal paths do not include a logic hazard, the reasonable choice of a method of modelling asynchronicity is *delaying clock domain output* since it is quite efficient (in contrast to the method of extending all critical input ports) and provides more refined models than plain or one-step destabilizers.

Chapter 4

Verifying Parametrized Hardware Designs via Counter Automata

In this chapter, we propose a novel way of verifying parametrized hardware components. Namely, inspired by the recent advances in the technology for verification of *counter automata*, we propose a translation from (a subset of) VHDL to counter automata on which formal verification is subsequently performed. The proposal was originally published in [93]. The subset of VHDL that we consider is restricted in just a limited way, mostly by excluding constructions that are anyway usually considered as erroneous, undesirable, and/or not implementable (synthesizable) in hardware.

In the generated counter automata, bit variables are kept track in the control locations whereas integer variables (including parameters) are mapped to (unbounded) counters. When generating counter automata from VHDL, we first pre-process the input VHDL specification in order to simplify it (i.e., to reduce the number of the different constructions that can appear in it), then we transform it to an intermediate form of certain behavioural rules describing the behaviour of particular variables that appear in the given design, and finally we put the behaviour of all the variables together to form a single counter automaton.

We have implemented a prototype tool performing the proposed translation. Despite there is a lot of space for optimising the generated counter automata and despite the fact that reachability analysis of counter automata is in general undecidable [69], we have been able to verify several non-trivial properties of parametrized VHDL components, including a real-life component implementing an asynchronous queue designed within the Liberouter project [80, 53].

In Section 4.3, we introduce some basics of VHDL, we comment on the VHDL constructions that we do not support, and explain the way we pre-process VHDL for the further transformations. We also introduce the notion of counter automata. In Section 4.4, we provide a translation from (simplified) VHDL to a certain form of intermediate behavioural rules. In Section 4.5, we present a translation from the in-

intermediate format to counter automata. In Section 4.6, we discuss our experimental results, and finally, in Section 4.7, we give a summary of the presented method.

4.1 Related Work

Recently, there have appeared many works on automatic formal verification of counter automata or programs over integers that can also be considered as a form of counter automata (see, e.g., [60, 62, 64, 68, 83]). In the area of software model checking, there have also appeared works that try to exploit the advances in the technology of verifying counter automata for a verification of programs over more complex structures, notably recursive structures based on pointers [61, 63, 66, 86].

In this chapter, we get inspired by the spirit of these works and try to apply it in the area of verifying generic (parametrized) hardware designs. We obtain a novel, quite general, and highly automated way of verification of such components, which can exploit the current and future advances in the technology of verifying counter automata (e.g., in [84], the authors were inspired by our counter automata models). To the best of our knowledge, there is no approach that provides verification of a generic HDL module for all of its possible instances¹.

4.2 Counter Automata

For an integer arithmetic formula φ , let $FV(\varphi)$ denote the set of free variables of φ .² For a set of variables X , let $\Phi(X)$ denote the set of integer arithmetic formulae with free variables from $X \cup X'$ where $X' = \{x' \mid x \in X\}$ (we assume $X \cap X' = \emptyset$). If $\mathbf{v} : X \rightarrow \mathbb{Z}$ is a valuation of variables from a set $X \supseteq FV(\varphi)$, we denote by $\mathbf{v} \models \varphi$ the fact that \mathbf{v} is a satisfying assignment of φ .

A *counter automaton* is a tuple $A = \langle X, Q, q_0, \varphi_0, \rightarrow \rangle$ where X is a finite set of counters, Q is a finite set of control locations, $q_0 \in Q$ is a designated initial location, φ_0 is an arithmetic formula such that $FV(\varphi_0) \subseteq X$, describing an initial assignments of the counters, and $\rightarrow \subseteq Q \times \Phi(X) \times Q$ is a finite set of transition rules. Instead of $(q, \varphi, q') \in \rightarrow$, we will normally write $q \xrightarrow{\varphi} q'$.

A configuration of a counter automaton is a pair $\langle q, \mathbf{v} \rangle \in Q \times \{\mathbf{v}' \mid \mathbf{v}' : X \rightarrow \mathbb{Z}\}$. The set of all configurations is denoted by \mathcal{C} . The transition relation $\xrightarrow[A]{\varphi} \subseteq \mathcal{C} \times \mathcal{C}$ is defined such that $(q, \mathbf{v}) \xrightarrow[A]{\varphi} (q', \mathbf{v}')$ iff there exist a transition $q \xrightarrow{\varphi} q'$ and a valuation

¹An anonymous reviewer of our work in [93] wrote: “...my first reaction was that someone has probably have done it, but googling ‘vhdl counter automata’ I got only the work of the authors.”

²We do not further restrict the kind of integer arithmetic used. It naturally follows from the integer operations used in the hardware design being handled, to which our translation adds just an implementation of the implicit modulo arithmetic used in VHDL—we will get back to this issue in the next subsection.

σ of $FV(\varphi)$ where $\sigma \models \varphi$, $\sigma(x) = v(x)$ for $x \in FV(\varphi) \cap X$, $\sigma(x') = v'(x)$ for $x' \in FV(\varphi) \cap X'$, and $v(x) = v'(x)$ for all variables $x \in X$ with $x' \notin FV(\varphi)$.

Just for a complete picture of a counter automaton, we denote by \xrightarrow{A} the union $\bigcup_{\varphi \in \Phi} \xrightarrow{A}^{\varphi}$, and by \xrightarrow{A}^* the reflexive and transitive closure of \xrightarrow{A} . A *run* of A is a sequence of configurations $(q_0, v_0), (q_1, v_1), (q_2, v_2) \dots$ such that $(q_i, v_i) \xrightarrow{A} (q_{i+1}, v_{i+1})$ for each $i \geq 0$ and $v_0 \models \varphi_0$.

4.3 Preprocessing Hardware for Translation to Counter Automata

Languages like VHDL or Verilog are simple to use for hardware design but they are very rich languages for direct translation to counter-automata-based model. In this section, we discuss how to transform a hardware design to a uniform description using a small set of language constructions. In the following, we will, in particular, consider the VHDL language. Both VHDL and Verilog have almost the same expressiveness in the field of RTL design, but VHDL is a very strongly typed language, thus VHDL designs are somewhat cleaner. Moreover, VHDL has a rich syntax in which different constructs have the same semantics (we will discuss this in details on page 78). As other RTL hardware description languages use constructs similar to VHDL constructs described in this section, one can easily understand how to apply the transformation of a model in other RTL language.

In the following, we describe the transformation of all common VHDL constructs, namely, parallel process definitions, signal/variable assignments, if-then-else conditions, select-when/case-when statements, user-defined functions, entity configurations, etc. Note that not all constructs of the VHDL deal with describing a hardware design. For instance, some constructs like procedure declarations, text input/output operations, and time delay expressions are used for test-bench specifications and are ignored in the synthesis phase when constructing the hardware as they do not have an influence on the behaviour of the hardware. Thus, we also ignore such construct when translating a hardware design to a counter automaton.

4.3.1 The Considered Features of VHDL

Due to the high complexity of VHDL, we do not cover all forms of VHDL constructs and all possible behaviours of the designed hardware. However, most of the restrictions that we describe below correspond to constructions or behaviour which are in theory possible, but are usually not used, represent undesirable design practices, or are often not even synthesisable. In particular, we do not consider cyclic assignments, processes sensitive on signal edges of two or more signals, or unstable states. Moreover, we restrict a hardware design to be used with some of IEEE library extensions

only, the scope of the parameters, and procedural features of VHDL.

Cyclic Assignments

We disallow cyclic assignments in the dataflow description or in the behavioural description of a latch. A cyclic assignment can be created in the dataflow description as a direct back-loop (cf. signal *q*):

```
q <= set or (switch xor q);
```

A cyclic conditional assignment within latches (cf. lines 4–5):

```
1 process(enable, a, b)
2 begin
3   if (enable = '1') then
4     a <= b;
5     b <= a;
6   elsif ...
7 end;
```

The first example describes the hardware in which the wire representing the signal *q* is directly connected to the logic which calculates the next value of *q*. The second example is similar in the way that the direct back-loop is through signals *enable*, *a*, and *b*. Such assignments would complicate our constructions significantly, and in practice, they are anyway undesirable as they lead to a possible oscillation of the signals (for the first example, this is the case of *set* equals 0 and *switch* equals to 1; for the second example, *enable* equals to 1 and values of *a* and *b* differs). Modern synthesis tools issue warnings when such constructions are used.

Sensitivity on More Clocks

A behavioural definition of a simple *process* (a process is a VHDL construct implementing a gate in a hardware design using the behavioural description), may contain a hierarchical if-then-else statement. A process referencing to an edge of a signal in an if-then-else statement (such a process is sensitive on an edge of such signal), would be most likely translated to a flip-flop. Since there is no flip-flop gate sensitive on two or more clock signals, we do not consider constructs like the following (cf. lines 3–4):

```
1 process(clk1, clk2)
2 begin
3   if (clk1'event and clk1='1') then
4     if (clk2'event and clk2='1') then
5       ...
6     elsif ...
7   elsif ...
8   end if; ...
9 end;
```

Unstable States

We concentrate on analysing reachable stable states of hardware components only (cf. Figure 2.6), thus we assume zero-delayed models only. In general, even when we are interested only in stable states, if we do not consider unstable states at all, there is a risk that we will not capture flaws caused by reading and registering unstable values. Such a flaw can be caused either (i) by a signal path that is too long wrt. the clock frequency used, or (ii) by an asynchronous exchange of signals between two clock domains. However, the need to deal with the former issue is eliminated simply by taking into account the capabilities of standard synthesis tools. These tools automatically check that the delay arising in the longest signal path of a given circuit is safe wrt. the clock frequency used and that checking that a designed system is in a stable state before a synchronization event happens (such as the rising edge of a clock signal). The latter issue is a more complicated but we have discussed dealing with this issue already in Chapter 3. Hence, below, we do not consider unstable states any further.

IEEE Library Types and Functions

There are several IEEE standards which extend the RTL design in VHDL via VHDL *packages*. A VHDL package declares and defines types and functions which can dramatically improve the design in a specific area in terms of development time. Such packages can define multi-valued logic types and functions which are suitable for simulation and test-bench purposes, floating-point types and arithmetic functions for development of arithmetic units, or mathematical operations which help with generic designs, e.g., the LOG2 function for calculating the bit vector width.

We do not consider several of the packages and IEEE recommendations such as floating-point types and arithmetics [70, 71], an extension for building tests [72], analog and mixed-signal extensions [74], constructions for controlling the process [76, 78], and timing specifications in a tabular form [77].

In the transformation, we also disallow mathematical packages [75]. Almost all arithmetic operations used from the mathematical package in the project [80], whose

source codes have been used for the case studies in this thesis, are used for calculation of the precise width of a bit vector. As we show below in the method of generating a counter automaton (Section 4.5.1), such information is not required.

The IEEE 1164 standard [79] defines non-binary logic types and functions over these types. Briefly, the `std_logic` and `std_logic_vector` types allow developers to define, tri-state logic circuits using the 'Z' high-impedance value, the undefined value 'X' (zero, one, high-impedance, ...), etc. Non-binary values of a wire are needed during simulation and test-bench phases for the analysis to see what is really happening in a modelled hardware. The place and route tool which maps the design to a particular hardware elements ignores these types. We also do not consider the package itself, but as these types are widely used in the RTL design, we abstract these types to binary types—`std_logic` is understood as `bit` and `std_logic_vector` as `bit_vector`. Arithmetic and relational operations over vector-typed variables implemented using a `for` cycle over items of a vector are transformed to their representations over base of a vector (mathematical $+$, $-$, $*$, unary $-$, $=$, \neq , \geq , ...). The arithmetic operation `abs` (the absolute value of a number) is disallowed as it complicates the transformation. Bit-wise logic operations (such as `and`, `or`, `not`, or `shl`) are the only ones which are considered from the IEEE 1164 package in the below transformations³.

Parameters of Generics

VHDL allows developers to write entities declared with parameters (so called generics) which can be instantiated several times with different values such that different parameter values cause different hardware representations.

In the transformation and analysis, we restrict integer parameters a bit (floating parameters are allowed). Namely, we do not allow a bit-wise access to variables with a parametric range and we do not allow `for` loops over parametrized variables (cf. Figure 4.1 and 4.2). Both of these restrictions could be lifted, but they would further complicate our translation to counter automata and also their analysis (as we would have to introduce a relatively complicated arithmetic formulae to mask out the particular bits of the values of particular counters). We let experiments with these feature for our future research.

Procedures and Functions

In VHDL, a developer can specify some behaviour via sequential statements in procedures and functions. We do not allow procedures and functions because they com-

³These functions are defined as a bit-wise access over one-bit elements along the whole length (width) of input vectors. As described bellow, such an access is in some circumstances disallowed, but their general denial would dramatically limit set of designs that the analysis could handle because it is common to use such a type of operations.

```

1  library ieee;
2    use ieee.std_logic_1164.all;
3    use ieee.std_logic_arith.all;
4  entity arith_unit is
5    generic ( WIDTH : integer = 64 );
6    port(
7      clk : in std_logic;
8      reset : in std_logic;
9      opcode: in std_logic_vector(0 to 1);
10     a, b: in signed(0 to WIDTH-1);
11     c: out signed(0 to WIDTH-1);
12     possible_carry : out std_logic;
13   );
14 end entity arith_unit;
15 architecture my_unit of arith_unit is
16   signal a1, a2, n1, n2: signed(0 to WIDTH-1);
17 begin
18   a1 <= a + b;
19   a2 <= a - b;
20   n1 <= -a;
21   n2 <= -b;
22   process(reset, clk)
23   begin
24     if (reset = '1')
25       c <= (others => '0');
26     elsif (clk'event and clk = '1') then
27       with opcode select
28         c <= a1 when '00',
29           a2 when '01',
30           n1 when '10',
31           n2 when '11';
32     end if;
33   end process;
34   possible_carry <= not opcode(1);
35 end architecture;

```

Figure 4.1: A VHDL design of a simple parametrized arithmetic unit. The parameter of the entity is an integer value specifying the width of input/output values. On such design, our transformation can be applied as there is no loop with a parametric range or a bit-wise access to signals with a parametrized range. Signal types included from `ieee.std_logic_1164` are transformed to binary-value types only.

```

1  library ieee;
2      use ieee.std_logic_1164.all;
3      use ieee.std_logic_arith.all;
4  entity plus_unit is
5      generic ( WIDTH : integer = 64 );
6      port (
7          clk : in std_logic;
8          reset : in std_logic
9          a, b: in std_logic_vector(0 to WIDTH-1);
10         sum: out std_logic_vector(0 to WIDTH-1);
11         carry : out std_logic;
12     );
13 end entity arith_unit;
14 architecture my_unit of arith_unit is
15     signal s: std_logic_vector(0 to WIDTH-1);
16     signal c: std_logic_vector(0 to WIDTH);
17 begin
18     c(WIDTH) <= '0';
19     for i in 0 to WIDTH-1 loop
20         s(i) <= a(i) xor b(i) xor c(i+1);
21         c(i) <= a(i) and b(i) or
22             a(i) and c(i+1) or
23             b(i) and c(i+1);
24     end loop;
25     process(reset, clk)
26     begin
27         if (reset = '1')
28             sum <= (others => '0');
29             carry <= '0';
30         elsif (clk'event and clk = '1') then
31             sum <= s;
32             carry <= c(0);
33         end if;
34     end process;
35 end architecture;

```

Figure 4.2: A VHDL design of a simple parametrized arithmetic unit, which is inappropriate for the described transformation. The number of circuits generated by the loop at lines 19–24 is specified by an integer parameter. There is also an access to particular bits of the parametric signal *c* at lines 18 and 32.

plicate the analysis (use of procedure parameters, local variables, conditional statements, recursive calls, etc.). In the hardware represented by such a description, the statements are not evaluated one after another, but the consequence of the whole block of sequential statements is taken into account only. Widely used synthesis tools also forbid the use of complicated sequential statement blocks, thus there is a small number of VHDL sources which use procedure and function statements. They are in general used in simulation specification.

4.3.2 Simplifying VHDL Code

A developer is able to use VHDL constructs to describe the same hardware design in different ways. For example, in the low-level, a developer can use either a behavioural or a dataflow description to express the same design (cf. Figure 4.3). More complex cases of equivalent descriptions can be achieved by coding based on functions, variable assignments, conditional assignment statements, more complex typed signals, etc.⁴

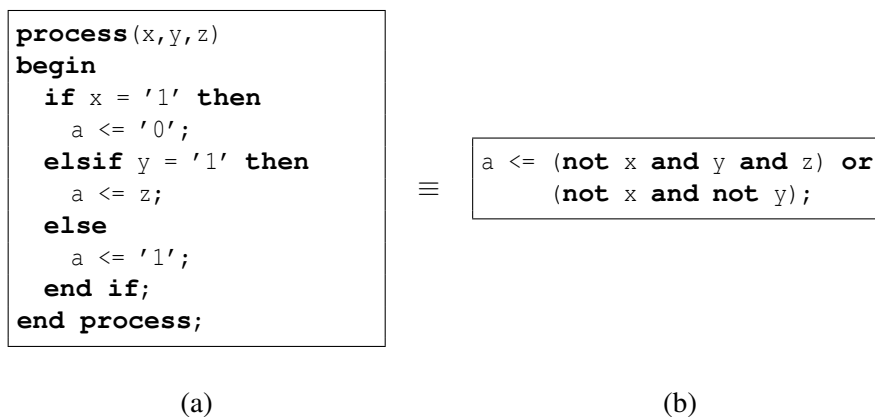


Figure 4.3: A simple example of two semantically equivalent circuits described using quite different syntactical means offered by VHDL. Sequential process shown in Part (a) is sensitive on signals x, y, and z which means that the process is triggered every time at least one of these signals changes its value. A new value of a is then calculated immediately. The dataflow description shown in Part (b) is, on the other hand, constructed using a logic expression. The difference between behavioural and dataflow description is in the syntax only.

To avoid a complex direct transformation from the VHDL language to counter automata, we first transform a VHDL source code to a form which is much simpler

⁴A difference can arise after the synthesis process when two equivalent constructions can lead to different implementations, thus to different timing characteristics. Fortunately, as we consider stable states only, timing characteristics are insignificant for us.

for all the subsequent transformation steps. The goal of the simplification is to obtain code consisting of conditional signal assignments only. We will show below that a set of conditional assignments can be used to replace any other VHDL statements, while preserving its semantics. The simplification consist of these steps:

1. Transforming structural descriptions to data-flow and behavioural descriptions.
2. Type simplification.
3. Transforming behavioural and dataflow description of a conditional statements to a uniform statement.
4. Normalization of conditional assignment statements.

We will describe each of steps of simplification in special section.

Transforming Structural Descriptions

The goal of the transformation is to obtain code describing only the behaviour or data-flow of the verified component.

We remove all structural descriptions of circuits and replace them by the corresponding behavioural description (in the way similar to macro expansion in the C programming language). This can easily be achieved by unfolding (or flattening) of the structural description taking into account that a structural description simply describes:

- from which subcomponents a given component is built of,
- what the values of the parameters of the subcomponents are, and
- how to connect the input/output ports of these subcomponents to the component.

The structural description is defined in a hierarchical way. When transforming a structural description to its behavioural representation, we transform it iteratively in a bottom-up way (the most nested subcomponent is transformed first). The transformation of a subcomponent (knowing its connection to the superior component) is done by substituting the instance of a subcomponent with its behavioural description and by substituting the references to ports of a subcomponent by signals of superior component connected to the instance of a subcomponent. If the subcomponent is an instance of a generic entity, its parameters are substituted by the constants from the subcomponent instantiation⁵. Also when injecting the behavioural description of a subcomponent into the component, we need to be beware of collisions of identifiers of signals. This can be solved by adding a unique prefix or suffix to one of the identifiers which mutually collide.

⁵When instantiate generic entities, all parameters must be set.

Type Simplification

As we mentioned before, VHDL components contain input/output ports, parameters, and internal signals and variables. VHDL provides two basic types of signals and variables: *1-bit (boolean)* and *arrays (vectors) of bits*. Further, there is also a possibility of user-defined structured types, but they are used as a form of syntactic sugar only. Therefore, before any further steps, we decompose structured signals and variables to their elements. Similarly, if a bit vector is accessed bit-wise (i.e., there is at least one statement in the considered code that accesses single bits of the vector at a time), we replace the vector with its boolean components (cf. Figure 4.4). If we had not disallowed the bit-wise access to parametrized-size vectors, we would have had to use a complex arithmetic formula to mask out the particular bits—e.g., to get a bit value at a position p in the bit vector represented by an integer value n , we could use the expression $(n \text{ div } 2^p) \bmod 2$, but such an expression would complicate the analysis of a counter automaton. The remaining vectors may then easily be mapped to counters of counter automata (whereas all 1-bit signals will be a part of their control states).

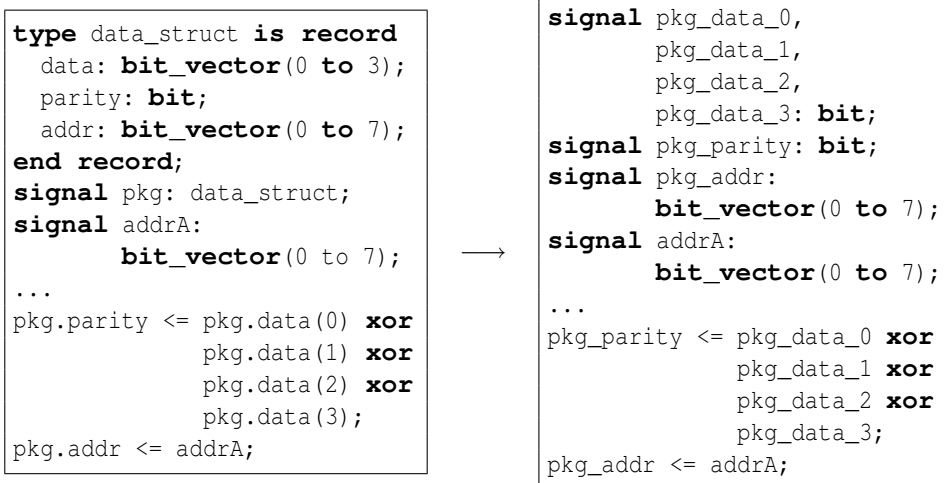


Figure 4.4: An example of simplification of an array and a structural type. Note that an array type is simplified only if its items are referenced (compare `pkg.data` with `pkg.addr`).

Transforming Behavioural and Dataflow Descriptions

Most of the behavioural code is written inside `process` statements. A process statement defines an independent sequential process representing the behaviour of some part of the design—how to change a value of one or more signals in dependence on other signals

We transform the code of a behavioural description such that the only statements that will remain (and that we will have to consider in the further steps) are the following:

1. *Assignment statements* of the form `signal <= expression;` appearing in an architecture definition as parallel statements or in a process statement as sequential statements.
2. *Conditional (if) statements* appearing in process sections as sequential statements with the following syntax (and the obvious semantics): `if cond1 then stmt1; elsif cond2 then stmt2; ... ; else stmtN; end if;`

Hence, one or more assignments and/or conditional statements are to be used to replace (without a change in the semantics) each occurrence of the following statements:

1. Selected assignments (*with-select* statements), i.e., a conditioned assignment of several values to one signal (cf. Figure 4.5a). An expression before the `when` keyword determines whether a value after the keyword should be assigned to a signal.
2. Sequential `case` statements, which are similar to selected assignments but in contrast to them, they can occur within process statements (cf. Figure 4.5b where a value of `v1` should be assigned to a signal `sig` if a value of signal `sel` equals to `c1`).
3. Generating `for` loops, i.e., statements which describe some repetitive parts of a design in an iterative way. In particular, loop generation statements are suitable for cases when the same operation has to be performed over an array-typed variable or signal for specific range of its items.

Given their semantics, the conversion of selected assignments and case statements to equivalent conditional statement is a straightforward exercise (illustrated in Figure 4.5) and we will omit its lengthy description here. It is also possible to get rid of the VHDL `for` loops by a simple loop unfolding mechanism (widely used in compiler optimization) since we assume that they cannot be performed over parametric bit vectors—otherwise, we would have to model their effect by special purpose loops in our counter automata.

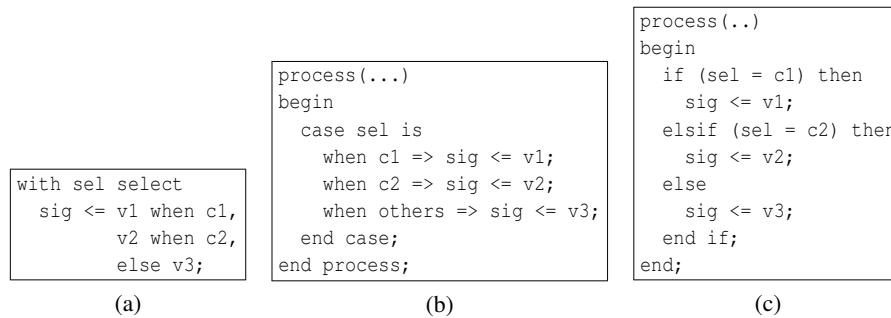


Figure 4.5: A illustration of the conversion of (a) *selected signal assignments* and (b) *case statements* to (c) *if statements*.

4.3.3 A Normalization of Conditional Assignment Statements

A key issue to be handled within the further transformation of VHDL towards counter automata is related to processes and the sequences of statements from which they are built. These statements are *not* executed sequentially—instead, for each variable, the last applicable assignment is searched and used, and all the statements preceding it are ignored. For example, for a sequence:

```
v <= e1;
if c then
    v <= e2;
end if;
```

if *c* holds, one performs the *v <= e2;* assignment, otherwise one performs the assignment *v <= e1;* (we may assume that the processes consist solely of assignments and (possibly nested) *if* statements).

In order to make dealing with the described semantics easier, we first transform each process into a single nested *if* statement in which it is clear under which conditions which assignment is to be applied. In particular, the example we mentioned above will be transformed to the statement:

```
if c then
    v <= e2;
else
    v <= e1;
end if;
```

More precisely, for each sequential process and each variable *v* assigned by that process, we do the following steps (we ignore all assignments to other variables when

handling v):

1. We add an empty `else` branch to each `if` statement of the given process that does not have such a branch.
2. Till there is some assignment or `if` statement s_1 in the given process that is just before an `if` statement s_2 (i.e., s_1 and s_2 are on the same level of nesting of `if` statements), we move s_1 to the beginning of the `else` branch of s_2 , i.e., we nest s_1 into the `else` branch of s_2 and put it just before the statements that are already in this branch.
3. If there are branches of `if` statements of the given process that do not contain any statement (either by ignoring assignments to other variables than v or by creating an empty `else` branch), we add the implicit assignment $v \leftarrow v$; to each of them.
4. We reduce every sequence of statements $s_1; s_2; \dots; s_n; v \leftarrow e_i$ within the given process to just $v \leftarrow e_i$. Here, s_i for $1 \leq i \leq n, n \geq 1$, is a sequence of assignments or `if` statements. The fact that at the end of the sequence there is an assignment statement (and not an `if` statement) is guaranteed by the transformation done in the previous step.

An example of the application of these steps is depicted in Figure 4.6.

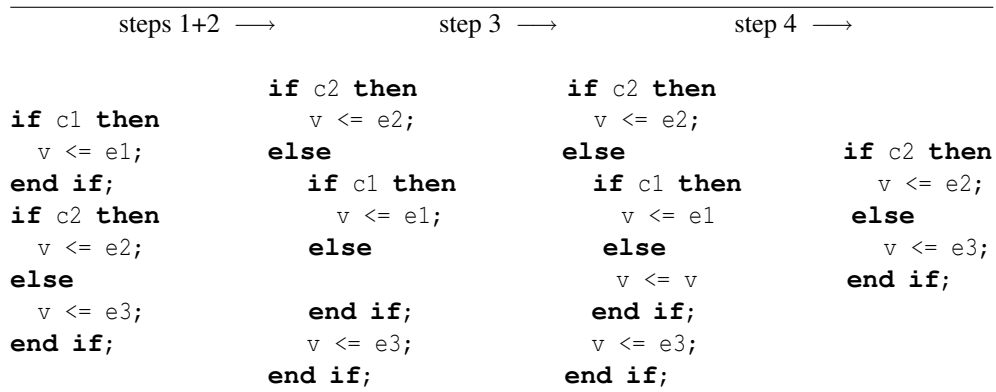


Figure 4.6: An example of transformations of sequential `if-then-else` statements.

Removing Variables

Another important issue when dealing with a normalization of conditional assignments are *variables*. Within a process definition in VHDL, variables and signals

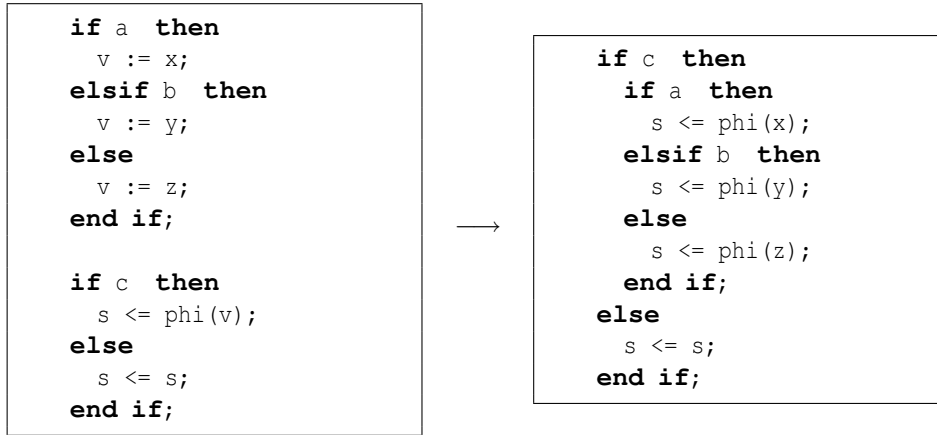
have different meanings. Signals most likely (depending on an optimization during synthesis) represent wires in a circuit. Variables, on the other hand, help a developer to express complex calculations of a signal value. A variable holds a result obtained from conditional assignments which can be used within calculations of other variables or signals. In other words, a variable serves as a temporary data storage for subsequent calculations of signals. For further analysis, we do not allow variables in a hardware design, but it is possible to remove them by the following method, preserving the same behaviour.

For the following description, we assume that previously described modifications of conditional assignments were applied both on signals and on variables. We will use *syntax trees* of such `if-then-else` statements representing a calculation of a single variable or signal. The nodes of a syntax tree denote the boolean expressions of *conditions* (i.e., the expressions between `if/elsif` and `then` keywords), branches from a node represent if a condition holds or not, and leaves denote the expressions on the right side of the assignment statements of a given `if-then-else` statement. We will use $\varphi(v)$ to denote an integer or boolean expression that has one or more references to a variable v . Further, let $\varphi[x/v]$ denote the expression $\varphi(v)$ in which every occurrence of v is substituted by x (cf. Figure 4.7a and Figure 4.7b for an illustration of a syntax tree).

In the following paragraphs, we will describe modifications of conditional assignment statements to remove all references to variables. We will perform the modifications on every sequence of conditional assignments in a top-down way since the sequential statements of a process is calculated in the same direction. By removing a variable, we must treat with variable references in two different ways depending on the place where the references occur: (i) a variable reference occurs within an expression which is assigned to a target signal, or (ii) a variable is used to express a condition of an `if-then-else` statement:

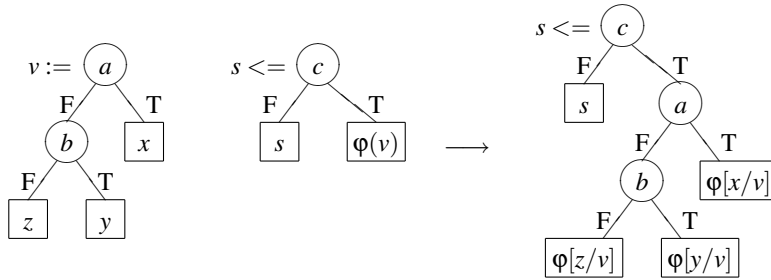
Variable references within assignments. If the variable v occurs in an expression on the right side of an assignment of some signal s , to get rid of v , we must replace the reference of v with its calculation. We operate over two syntax trees: the syntax tree t_v of a calculation of v and the syntax tree t_s of calculation of s . We substitute every leaf $\varphi(v)$ of t_s with the syntax tree t_v in which every leaf e is substituted with $\varphi[e/v]$. An example of such a transformation is depicted in Figure 4.7.

Variable references within conditions. If the variable v of the syntax tree t_v occurs in a condition of the conditional assignment of signal s of the syntax tree t_s , the substitution is a bit more complex than the previous reference removal. We need to substitute the whole condition with regard to all possible values of the variable. We substitute every condition $\varphi(v)$ of t_s with $\bigvee_{\phi \in P_v} \text{conj}_{\phi}$. Here, P_v is a set of all paths of t_v from the root node to the leaves. Further, conj_{ϕ} stands for the conjunction



(a)

(d)



(b)

(c)

Figure 4.7: An example of removing variable references from an expression as a value being assigned to a signal or a variable: (a) original if-then-else statements setting the variable v and the signal s , (b) syntax tree representing the conditional assignment statements, (c) syntax tree with substituted references of the variable v , and (d) syntax tree transformed back to VHDL language.

of all conditions along the path ϕ and is defined as $conj_{\phi} = \bigwedge_{n \in \phi} e(n)$ where $e(n)$ is the boolean expression of the node n defined as follows:

1. If n is a leaf representing some expression x , then $e(n) = \phi[x/v]$.
2. If n is a node representing some condition c and the next node in the path

p follows the right branch of the node n , i.e., meaning that the condition is satisfied, then $e(n) = c$.

3. If n is a node representing some condition c and the next node in the path p follows the left branch of the node n , i.e., meaning that the condition is *not* satisfied, then $e(n) = \neg c$.

An example of the just described modification is depicted in Figure 4.8.

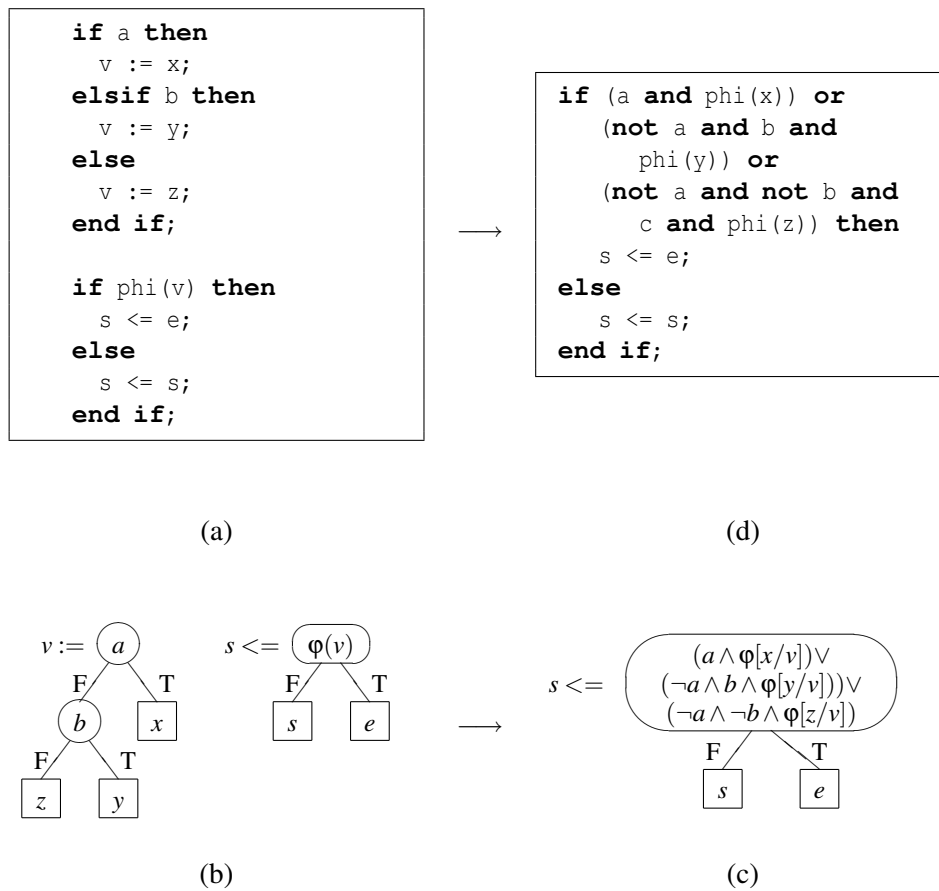


Figure 4.8: An example of removing variable reference from boolean expression in the condition part of if-then-else statement: (a) original if-then-else statements setting the variable v and s , (b) syntax tree representing the conditional assignment statements, (c) syntax tree with substituted references of variable v , and (d) transformed substituted syntax tree to VHDL language.

4.3.4 Handling VHDL Integer Variables in Counter Automata

When translating operations on integer variables used in VHDL to operations on counters, we have to take care of the fact that in VHDL, arithmetical operations over integers are always implicitly evaluated *modulo the range of the appropriate integer variables*. In counter automata, we have to make the modulo computation explicit (e.g., an assignment $v_1 \leftarrow v_2 + v_3$; over integer variables represented on n bits has to be translated to an assignment of the form $v_1 := (v_2 + v_3) \bmod 2^n$).

For analysing the generated counter automata, we then, of course, need a tool that can cope with counter manipulations corresponding both to arithmetical, logical, and relational operators directly used in the considered VHDL design as well as to the additional operations stemming from implementing the implicit modulo computations (and if we add them in the future, then also the bit-wise manipulations on integer variables). Given a concrete counter automata analyser, the translation may need to be adjusted to respect the operations that the tool supports. If the tool does not offer all the needed operations (nor allows their implementation based on other supported operations), one has to restrict to the case when the appropriate integer variables have a fixed range (i.e., are not parameters) and can also be recorded as a part of the control states of counter automata, or, alternatively, try to find some approximation of the missing operations based on what is available (in the worst case using purely random operations).

4.4 An Intermediate Behavioural Model

In the previous section, we discussed the syntax and semantics of VHDL constructions that we will consider in the following, together with the notion of counter automata that we want to use to model (and analyse) these constructions. In order to make the translation from the simplified VHDL to counter automata smoother, we implement the translation via an intermediate behavioural model that we will now present.

4.4.1 A Definition of the Intermediate Behavioural Model

The *intermediate behavioural model* of a hardware component is defined as a triple $M = (V, T, B)$ where:

- V is a finite set of variables,
- $T : V \rightarrow \{\text{bool}, \text{int}\}$ is a function that associates every variable with the boolean or the integer type, and
- B is a finite set of *behavioural rules* that describe the behaviour of a given hardware component and that have a form which we introduce below.

Let $V_i \subseteq V$ be a set of input ports and $V_p \subseteq V$ a set of parameters. We define $\bar{V} = V \times \{\text{last}, \text{next}, \text{posedge}, \text{negedge}\}$ to be the set of possible references to the values of variables from V with the following meaning:

- $(v, \text{last}) \in \bar{V}$ refers to the value of v in the *last reached* (i.e., current) state—in expressions, we usually abbreviate it simply to v ,
- $(v, \text{next}) \in \bar{V}$, abbreviated to v' , denotes the value of v in the *next state*,
- $(v, \text{posedge}) \in \bar{V}$, abbreviated to $\uparrow v$, has the boolean meaning $\uparrow v = \neg v \wedge v'$ and denotes the *positive edge* of a 1-bit variable v (for which $T(v) = \text{bool}$),
- $(v, \text{negedge}) \in \bar{V}$, abbreviated to $\downarrow v$, has the boolean meaning $\downarrow v = v \wedge \neg v'$ and denotes the *negative edge* of a 1-bit variable v (for which $T(v) = \text{bool}$).

Further, let E be the set of all (well-typed) expressions that one can form over \bar{V} using arithmetical ($+$, $-$, $*$, \dots), relational ($=$, \neq , $<$, $>$, \leq , \geq), and logical (\neg , \wedge , \vee , \dots) operators, and let C be the subset of E containing all boolean-valued expressions. Let $\perp \in E$ denote an *empty* expression (see below).

We can now introduce special conditional assignments that play the role of the behavioural rules constituting the set B of an intermediate behavioural model. In particular, $B \subseteq C^* \times V \times E$. We write a behavioural rule $b \in B$ as

$$c \rightarrow v := e$$

for $c \in C^*$ being a list of enabling conditions, $v \in V$ the variable set by the rule, and $e \in E$ being an expression defining the new value of v . In other words, a behavioural rule b with a list of enabling conditions $c = c_1 c_2 \dots c_n$ says that if $c_1 \wedge c_2 \wedge \dots \wedge c_n$ holds for a given valuation of the variables, v will get a new value obtained by a valuation of e . If $c = \epsilon$, we consider it to be always true, and the assignment $v := e$ is always enabled.

For a behavioural rule $b : c \rightarrow v := e \in B$, we denote:

$$\text{cond}(b) = c \text{ the enabling condition of } b,$$

$$\text{var}(b) = v \text{ the variable to be set,}$$

$$\text{value}(b) = e \text{ the expression defining the new value of } v.$$

Further let,

$$F(e), \text{ for } e \in E \cup C^*, \text{ be the set of reference to variables occurring in } e, \text{ and}$$

$$B(v) = \{b \mid b \in B, \text{var}(b) = v\} \text{ be the set of behavioural rules over a variable } v.$$

4.4.2 Extracting Behavioural Rules from the Source Code

The architecture of a VHDL component consists of a set of parallel assignments and a set of sequential processes. With respect to the simple VHDL transformations described in Section 4.3.2, we may assume that the sequential processes consists of a single `if` statement for every variable set within it. In order to obtain the set of behavioural rules B from such a description, we extract the rules from VHDL statements as follows:

1. For each parallel assignment $v \leq e_i$, we add a rule $\varepsilon \rightarrow v := e$ into B .
2. For each sequential process that sets a variable v by a single, possibly nested, `if` statement (after the pre-processing, there is no other possibility), we proceed as follows. For each assignment statement $v \leq e_i$ that appears on the leaf level of such a (nested) `if` statement, we add a rule $c'_1, c'_2, \dots, c'_n \rightarrow v := e$ into B ($n \geq 1$). Here, c_1, c_2, \dots, c_n are all the branching conditions that one tests before reaching $v \leq e$, and $c'_i = c_i$ if the condition is supposed to hold (i.e., we are nesting into an `if` c_i or `elsif` c_i branch) whereas $c'_i = \neg c_i$ if the condition is supposed not to hold. An example of such a transformation is shown in Figure 4.9.

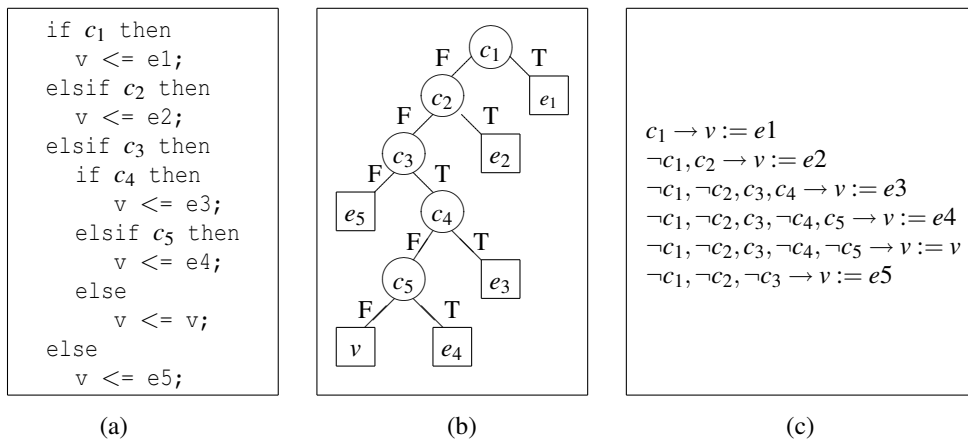


Figure 4.9: Synthesis of behavioural rules wrt. the conditions passed till a certain assignment can be fired: (a) a normalized VHDL `if` statement, (b) the syntax tree representing the `if` statement, (c) the set of behavioural rules for the variable v derived from the given `if` statement.

4.4.3 Adjustments of Behavioural Rules

The environment of a component

To be able to model check a component, we need a model of its environment too. Currently, we model the environment to behave in a completely random way. To do that, we extend the intermediate behavioural model by adding behavioural rules for all component inputs. For every such an input $v \in V_i$, we add the following behavioural rules:

- If v is a 1-bit variable (i.e., $T(v) = \text{bool}$), we add the rules $\varepsilon \rightarrow v := v$ and $\varepsilon \rightarrow v := \neg v$ to B . In this case, we tolerate that these two rules have a conflicting enabling condition.
- Otherwise (i.e., $T(v) = \text{int}$), we extend B by the rule $\varepsilon \rightarrow v := \text{random}$. Here, *random* represents a random integer value. Note that we have to adjust the form of *random* such that the counter automaton analyser that we want to use understands it.

Non-state variables

We are only interested in stable states that are defined by the so-called *state variables*. In the hardware developers' jargon, such variables are also known as registers or signals which save their value. The remaining variables are *non-state variables* whose values are not registered and that, from our point of view, represent just a symbolic name for some expression. From a set of behavioural rules, a non-state variable can be identified by the fact that its value is set by a rule with the empty enabling condition (i.e., by an unconditional assignment⁶). The remaining variables are then state variables. The only exception are input variables whose values are defined and held by the environment of the modelled component. Formally, $v \in V \setminus V_i$ is a non-state variable iff $\text{cond}(b) = \varepsilon$ for the rule $b \in B$ such that $B(v) = \{b\}$. Let further $V_s = V_i \cup \{v \mid v \in V, \text{cond}(b) \neq \varepsilon\}$ be the set of state variables. Before generating counter automata, we change the intermediate behavioural model to use the state variables only. We remove the non-state variables v defined by rules $\varepsilon \rightarrow v := e$ present in B by iteratively searching for references to such variables in enabling conditions and value expressions of the rules in B and by replacing these references by e .

Behavioural rules over 1-bit variables

Next, for technical reasons allowing us to ease the subsequent construction of a counter automaton from intermediate behavioural rules, we prefer to have all the manipu-

⁶Note that as we require the rules not to be in a conflict, this is the only rule that is setting the value of such a variable.

lation of 1-bit state variables in guards of the rules. That is why, we transform every behavioural rule $b : c \rightarrow v := e$ over a 1-bit state variable $v \in V_s, T(v) = \text{bool}$, to the rule $b_{\text{new}} : c, v' = e \rightarrow v := e$.

Triggers of behavioural rules

Let $V_{\uparrow\downarrow} = \bar{V} \cap (V \times \{\text{posedge}, \text{negedge}\})$ be the set of edges of the values of variables from V . We define a mapping $R : B \rightarrow \{\tau\} \cup V_{\uparrow\downarrow}$ that assigns to each rule either τ in case the rule models an assignment in the transparent mode or a signal edge (i.e., a *trigger*) that activates the rule if it models an assignment in the synchronous mode. Formally, for $b \in B$, let $R(b) = \tau$ iff $F(\text{cond}(c)) \cap V_{\uparrow\downarrow} = \emptyset$, and let $R(b) = t$ iff $F(\text{cond}(b)) \cap V_{\uparrow\downarrow} = \{t\}$ for some $t \in V_{\uparrow\downarrow}$. Note that this definition is correct since due to the hardware description principles, there can be at most one positive or negative edge variable reference in a behavioural rule condition. Designs violating this requirement are exposed during the synthesis process.

For each rule $b \in B$ that works in the transparent mode, i.e., $R(b) = \tau$, we adjust the condition and assignment part of b such that each variable reference that appears there refers to the future. This is, we change every variable reference v that appears in $\text{value}(b)$ or $\text{cond}(b)$ to v' . The reference to the future assures that the rule is evaluated using values of variables that are computed at the same time step as the one at which we perform the valuation (and not a step before as in the case of the synchronous mode). This is because gates working in the transparent mode immediately propagate their input values to the output. We can afford to use this transformation as we excluded the possibility of cyclic dependencies of the values of variables in the transparent mode. That is why, the variables changing in the transparent mode can be ordered according to their dependencies and evaluated in the given order starting with variables that are assigned a constant value (which happens, e.g., when the circuit is being reset) or from variables which are not changing at the given time step. For an illustration of this behaviour, see Figure 4.10.

We have to do a similar adjustment as above also for the rules modelling the synchronous mode (cf. Figure 4.11). For simplicity, we consider here the case of positive edges only. The case of negative edges is analogical. Within each rule $b \in B$ for which $R(b) = \uparrow v$ for some $v \in V$, $\text{cond}(b) = c_1 c_2 \dots c_n \uparrow v c_{n+1} \dots c_m$ for some $n, m \in \mathbb{N}$. Note that $F(c_1 c_2 \dots c_n) \cap V_{\uparrow\downarrow} = \emptyset$. In this case, the way our algorithm for generating behavioural rules works implies that the set of generated behavioural rules B must also include behavioural rules $b_\tau \in B$ whose condition is built solely of the conditions c_1, c_2, \dots, c_n (possibly negated), hence $R(b_\tau) = \tau$. Due to the order of the evaluating conditions, the b_τ rules have a priority over b . At the same time, they model the transparent mode, hence they will work with the future values of variables. That is why, in order to exclude a possible conflict of the rules b_τ with b , we have to replace every variable reference v to v' in c_1, c_2, \dots, c_n in b . Then, if some of the b_τ rules is enabled, b is disabled as its enabling condition contains a negation of

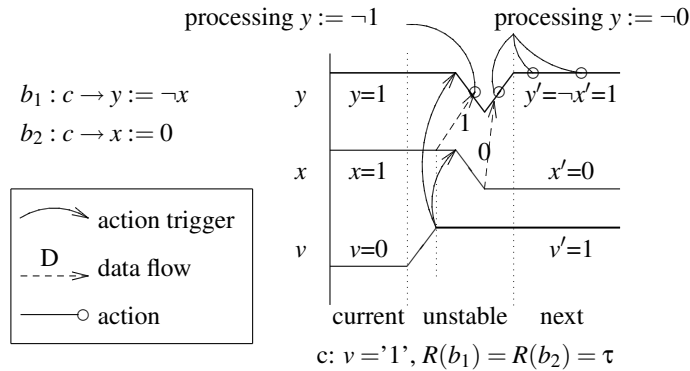


Figure 4.10: A timing diagram illustrating conditional assignments of values to signals in the *transparent mode*. Both x and y are controlled by the *level* of the variable v , which causes a continuous change of their values (y is set to the negation of x via b_1 , y is set to 0 via b_2). Due to the propagation delays of hardware which implements such a behaviour, there are several changes of the values until they are all stabilized, which we are, however, not interested in. The important thing to notice is that the resulting value of x is $\neg y'$ and not $\neg y$.

some of the enabling conditions of b_τ evaluated on the same values of variables. On the other hand, if this is not the case, the rest of b will work with the current values of the variables. An illustration of the construction is depicted in Figure 4.12.

4.5 Generating Counter Automata

In previous modifications, we prepared a set of behavioural rules representing the behaviour of a generic component and its environment. In the following, we will describe how to transform these rules into a counter automaton.

4.5.1 Counters, Control Locations, and Initialization

Counters. Let us fix a hardware design with a set of variables V of types T and with a set of behavioural rules B generated from the design. We start building the counter automaton A representing the design by defining its *set of counters* as the set of all integer-typed state variables from V —formally, wrt. the definition of counter automata (Section 4.2), $X = \{v \mid v \in V_s, T(v) = \text{int}\}$.

Control locations. Further, we build control locations of A based on all possible valuations of all *control state variables* in V , i.e., 1-bit state variables from the set

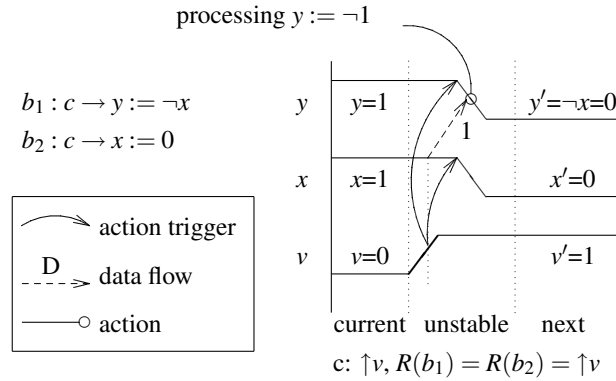


Figure 4.11: A timing diagram of conditional assignments of values to signals in the *synchronous mode*. In synchronous mode, an *edge* triggers a change of the value of v , which holds until the next triggering event. In this case, the resulting value of x is $\neg y$ (and not $\neg y'$). Note that the expression $\uparrow v$ is true iff $\neg v \wedge v'$.

$V_q = \{v \in V_s \mid T(v) = \text{bool}\}$. Formally, we define the set of control locations of A as $Q = \{q \mid q : V_q \rightarrow \{0, 1\}\}$.

Initialization. The design of a component in VHDL does not include any specification of its initial state. In most cases, however, the specification of the component includes a combination of signals which *resets* the component to some initial state and assigns some constants to all its internal variables. For the generation of A , to obtain these constants and thus define the *initial location* and the *initial constraint on counters*, the user must explicitly specify the resetting signals by providing the appropriate valuation of input variables that encodes them. By evaluating enabling conditions of all the rules in B under the given resetting valuation of the input variables, we get a subset of rules that are initially enabled. Each of such behavioural rules defines an initial value for one variable—by evaluating the assignment parts of these rules, we can initialize the variables. The obtained values of control state variables make up the definition of the initial location q_0 , the valuation of integer variables allows us to construct the initial constraint φ_0 on counters⁷. If the modelled component has no resetting signals or the desired initial state is not the reset state, the initialization must be defined explicitly by the user.

⁷In fact, this applies only to the counters other than the ones representing parameters—if the possible values of parameters are also to be constrained somehow, it is up to the user to add the appropriate constraint into φ_0 .

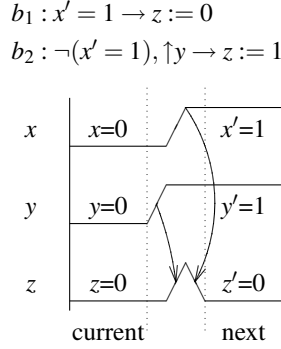


Figure 4.12: A timing diagram illustrating the interplay between rules working in the transparent and synchronous mode. Assume that the positive edge of y is faster than x , then z is controlled by the synchronous mode rule b_2 , but before a stabilization, the high-priority variable x forces z to be set transparently (wrt. the rule b_1).

4.5.2 The Transition Relation

For an expression $e \in E$ and two locations $q_1, q_2 \in Q$ of A , we denote by e^{q_1, q_2} the valuation of e where for each $v \in V_q$, $(v, last)$ is evaluated as $q_1(v)$ and $(v, next)$ is evaluated as $q_2(v)$. We allow the valuation to be partial—if e contains integer variables, they remain untouched. We construct the transition relation of A by checking for every pair of control locations $q_1, q_2 \in Q$, $q_1 \neq q_2$, whether the intermediate behavioural model allows us to connect them:⁸

1. For each $b \in B$ with $cond(b) = c_1 c_2 \dots c_n$ for some $n \in \mathbb{N}$, we (as far as possible) evaluate the enabling condition of b , i.e., we compute

$$guard^{q_1, q_2}(b) = \bigwedge_{1 \leq i \leq n} c_i^{q_1, q_2}$$

Further, let

$$B_e = \{b \mid b \in B, var(b) \in V_s, guard^{q_1, q_2}(b) \neq false\}$$

be the set of all (conditionally) enabled behavioural rules setting the value of state variables.

2. We further one-by-one consider all subsets $B_t \subseteq B_e$ such that B_t contains exactly one rule b such that $var(b) = v$ for each state variable $v \in V_s$. For each B_t , we perform the following steps:

⁸Note that we cannot have self-loops in A as the control states are stable, and some signal must change in order a change of the states happens.

- (a) In each rule $b \in B_t$, we iteratively substitute all references to the future values of counter variables by the expressions assigned to them within B_t . This is, we substitute each v' for $v \in V_s \setminus V_q$ by the expression $value(b_v)$ where $b_v \in B_t$ and $var(b_v) = v$.⁹ We repeat this step till all references to future values of counters disappear.
- (b) Based on the set of rules B_t , we create a transition $q_1 \xrightarrow{\varphi} q_2$ of A where

$$\varphi = \left(\bigwedge_{b \in B_t} guard^{q_1, q_2}(b) \right) \wedge \left(\bigwedge_{b \in B_t, var(b) \notin V_q} \alpha(var(b) := value^{q_1, q_2}(b)) \right)$$

and α is a function that transforms an assignment $v := e$ to a formula $v' = e$.

Let us add a few comments to the algorithm. For a given choice of states q_1 and q_2 , the first step may lead to three situations: (i) If $guard^{q_1, q_2}(b) = false$, we know that b does not change the value of $var(b)$. (ii) If $guard^{q_1, q_2}(b) = true$, b is allowed to change the value of $var(b)$. (iii) Finally, if $guard^{q_1, q_2}(b)$ does not reduce to neither *false* nor *true* (i.e., if $guard(b)$ refers to some values of counters in a way that must be taken into account), we only know that b may be able to change $var(b)$, but subject to the values of the counters. If there is no (at least conditionally) enabled behavioural rule for some state variable, i.e., if $\exists v \in V_s, \forall b \in B(v). guard^{q_1, q_2}(b) = false$, no transition from q_1 to q_2 will be possible as we are unable to compute the next value of v in q_2 —even for preserving the current value of v there is a behavioural rule which is forbidden by its guard. Otherwise, we have to explore all combinations of (at least potentially) enabled rules adjusting the value of the particular variables, which is done in the second step of the algorithm.

Suppose now that, for instance, $V_s = \{v_1, v_2, v_3, v_4\}$ where only v_4 is a 1-bit variable, and the first step of the algorithm yields a set of rules:

$$B_e = \{$$

$$\begin{aligned} g_1 &\rightarrow v_1 := f_1(v'_2), \\ g_{2,1} &\rightarrow v_2 := f_{2,1}(v'_3, v_1), \\ g_{2,2} &\rightarrow v_2 := f_{2,2}(v_3), \\ g_3 &\rightarrow v_3 := f_3(v_2), \\ v'_4 &= \neg v_4 \rightarrow v_4 := \neg v_4 \end{aligned}$$

⁹At this point, only the variables representing counters are considered since the references to future values of control state variables are taken care through the partial valuation of the expressions.

}
 (the rule for v_4 is transformed as we described in Section 4.4). We can find two subsets B_t that are to be handled by the second step of the algorithm—namely,
 $B_{t,1} = \{$

$$\begin{aligned} g_1 \rightarrow v_1 &:= f_1(v'_2), \\ g_{2,1} \rightarrow v_2 &:= f_{2,1}(v'_3, v_1), \\ g_3 \rightarrow v_3 &:= f_3(v_2), \\ v'_4 = \neg v_4 \rightarrow v_4 &:= \neg v_4 \end{aligned}$$

} and
 $B_{t,2} = \{$

$$\begin{aligned} g_1 \rightarrow v_1 &:= f_1(v'_2), \\ g_{2,2} \rightarrow v_2 &:= f_{2,2}(v_3), \\ g_3 \rightarrow v_3 &:= f_3(v_2), \\ v'_4 = \neg v_4 \rightarrow v_4 &:= \neg v_4. \end{aligned}$$

}

If we apply the steps described above for $B_{t,1}$, we obtain two counter-automata transitions with a formula

$$g_1 \wedge g_{2,1} \wedge g_3 \wedge v'_1 = f_1(f_{2,1}(f_3(v_2), v_1)) \wedge v'_2 = f_{2,1}(f_3(v_2), v_1) \wedge v'_3 = f_3(v_2)$$

going between control states q_1 and q_2 such that $q_1(v_4) = \neg q_2(v_4)$. Note that the condition $v'_4 = \neg v_4$ does not appear in the formula of the transition as its valuation wrt. q_1, q_2 yields *true*.

4.6 Experiments

For our experiments, we implemented in Python a prototype version of the translation that we proposed here (up to some issues of the VHDL pre-processing mentioned in Section 4.3), available online in [91]. In particular, we implemented a translation to counter automata in the input language of the ARMC tool [83] and also to integer programs in the C programming language for Blast [67]. Both of the tools are based on predicate abstraction and the CEGAR approach.

We have concentrated on verifying of safety properties, i.e., on *reachability of some bad states*. To distinguish the bad states, we change a given VHDL specification by creating a 1-bit variable whose value is a propositional logic formula representing the bad states. The translation then distinguishes states with this bit unset (the good ones) and set (the bad ones). Transitions to the bad states are controlled by the formula describing what a bad state is.

To test the proposed counter-automata-based model extraction method, we have first applied it to two small non-parametric components (having integer variables, but of a fixed width). Then we applied the method to two more complex parametric components, including a real-life, specialized parametric component developed within the Liberouter project [80].

The first two components (a counter and a register) represent basic elements from which hardware is built on the RTL level. For the *counter*, we verified that there is no overflow possible. For the *register*, we verified that the data transfer from its input to the output and the reset of the register function correctly. A more complex case study that we considered is a *synchronous LIFO* component which implements a stack with two operations—push and pop. The generic nature of this component is given by a parametrization of the number of items the LIFO can save. This component implements among other signals whether it is empty or full. We verified whether these signals are always set correctly for any possible size of the LIFO.

The last verified component is an *asynchronous queue* (FIFO). This specialized parametric component was built to be used in network monitoring adaptors developed within the Liberouter project (with a stress on being as efficient as possible). We successfully verified two properties of the component:

- The queue does not inform that *it is full* incorrectly. We introduced a counter of written items (`written`) for specifying the bad state formula (in VHDL):
`bad <= not reset and full and written<size-3;` i.e., if a component is working properly (`not reset`), then `full` should not be set if the number of written items is less than the size of a queue (`written<size-3`, here, three items are reserved for a delay of the synchronization between reading and writing clock domains).
- The queue does not inform that it is *empty and full at the same time*, i.e.,
`bad <= not reset and full and empty;`

We have verified the first property on four different configurations of the component (we modify the configuration of the FIFO by limiting its behaviour via its input signals). In particular, four configuration includes the case when (i) only the writing part of the FIFO was enabled (reading clock signal `clk_rd` set to 0), (ii) the whole FIFO was enabled, but no data reading was performed (read bit `rd` set to 0), (iii) the whole component was active, all operations except the reset was allowed (`reset = 0`), and (iv) all possible behaviours of the FIFO was verified.

Component	$ Q $	$ \delta $	$ X $	Extraction time	ARMC	Blast
Counter	6	14	2	< 1s	< 1s	3.4s
Register	10	44	2	1s	< 1s	2.6s
SynLIFO	66	985	2	23s	1.7s	8h
AsFIFO-Full (i)	18	205	7	2s	1s	N/A
AsFIFO-Full (ii)	66	838	12	45s	7s	N/A
AsFIFO-Full (iii)	66	2628	12	1m3s	1m21s	N/A
AsFIFO-Full (iv)	130	4148	12	3m42s	4h6m	N/A
AsFIFO-FE	34	612	11	16s	15h29m	N/A

Table 4.1: Experiments with extracting from VHDL and with their subsequent reachability analysis using ARMC and Blast.

When verifying the latter property, we also modify the behaviour of an asynchronous FIFO a bit. According to the specification of the FIFO, we must assure that both clocks (reading and writing) are active all the time and the following holds: $T_{slow}/T_{fast} \ll Size$ where T_{slow} is a period of a slow clock (reading/writing), T_{fast} is a period of a fast clock (writing/reading respectively), and $Size$ is the size of a queue. Since both tools we have used in our analysis of counter automata do not support fairness properties, we made the FIFO synchronous by making one clock signal to be a non-state variable holding a value of another clock signal, in VHDL, `clk_rd <= clk_wr;` (thus the whole component is sensitive on `clk_wr` only).

The results of our experiments are summarized in Table 4.1. The first column identifies the component and/or the verified property—Counter and Register as the basic blocks of RTL design, the component of a LIFO queue (SynLIFO), and two safety properties for asynchronous FIFO, one of which is performed on different configurations of the FIFO (verified properties were satisfied for all the cases). The next column provides the number of control locations in the counter-automaton model—note that the number corresponds to $2^n + 2$, which is the number of control locations over n 1-bit state variables, one location representing an initial state constraining values of parameters, and one location representing a bad state. Columns 2 and 3 represent the number of transitions between control locations and the number of used counters (integer variables). Column 4 represents the time needed for translating the designs into counter automata by our prototype. Columns 5 and 6 represent times needed for analysing the generated counter automata in ARMC and Blast (“N/A” means that the verification did not finish). The experiments were performed on an Intel E6750 processor with 2GB DDR3 memory with single-threaded ARMC version 3.20.05 (2003) and Blast version 1.0 (2003).

4.7 Summary of the Translation

The bottleneck of the presented method of translating a VHDL design to a counter automata is in generating a transition relation. The complexity of the construction is $O(mn^2)$ where m is the number of behavioural rules (in other words, it represents how complex is the behaviour of a component) and n is the number of one-bit state variables. The complexity is linear to m and quadratic to n since the construction deals with every behavioural rule for each pair of control states individually.

The time needed for a verification is influenced both by the verified property and the tool used for an analysis. The best results were provided with the ARMC checker since it was able to prove the correctness of every model (the problem is undecidable in general). The Blast was able to finish on non-complex components only, for others it reached a 24 hours limit. We also experimented with other counter-automaton analysers like Fast [62], Faster [87], or ASPIC [88], but the results provided with these tools were even worse than the results of Blast.

Despite the ARMC was able to finish in all our experiments, applying the method of verifying parametrized hardware designs proposed in this chapter is reasonable on small components only. Comparing with traditional simulation-based approaches, formal analysis of parametrized hardware designs is limited due to the restrictions of VHDL constructs described in Section 4.3.1 and it also contributes to state-space explosion problem due to the quadratic increase of the size of an automaton wrt. the number of bit-state variables. In comparison with formal verification of concrete, non-parametrized designs, the proposed approach lacks a possibility of uncomplicated definition of an environment of verified component (input signals can be either random or constant for an analysis). On the other hand, the proposed technique allows one to verify (in a limited scope) a parametrized hardware design for all possible configurations at once.

Chapter 5

Conclusion and Future Research

In the thesis, we have introduced two original approaches to formal verification of hardware designs. We conclude each of the methods in its own section.

5.1 Verifying Clock Domain Crossings

For verifying asynchronous systems with multiple clock domains, we have introduced four original approaches, two of which were published in [92], the two other ones are going to be published in the near future. All of the approaches refine the zero-delay modelling of hardware designs with an appropriate description of transient behaviour of a circuit. Such a refinement allows the quality engineer to perform a check of clock domain crossings within the functional verification of a circuit with no need of verifying CDC independently. The proposed approaches differ in their precision and the incurred verification cost.

The first of the proposed approaches (based on extending all critical input ports) is quite precise, but may contribute to the state explosion problem in a significant way since it introduces a number of new state variables. The other approaches are much more efficient as they are based on an over-approximation of the behaviour of the clock domain crossing, and hence use less new variables. The methods are, however, still precise enough to allow one to prove interesting properties on various hardware designs as we have illustrated by our experiments. Every method is implemented in the prototype tool *CDCreveal* [90], which uses the input language of the Cadence SMV model checker. The implementation can be simply modified to be used with a different model checker.

In summary, the proposed approaches represent a contribution to the state-of-the-art in verifying hardware by allowing one to deal with asynchronous circuits with no need of special care of clock domain crossings.

5.1.1 Future Directions in Verifying CDC

Because of the over-approximation of the possible behaviour that arises when using destabilizers, one-step destabilizers, or delaying gates on clock domain outputs, one cannot be sure if detected problem reflects a possible behaviour of a real system. Thus, one must verify the reason of such an alarm. This could be done either manually or by a more precise method (the only precise method proposed in the thesis is the extension of all critical input ports). Unfortunately, neither is really practical since manual intervention is undesirable and the extension of all critical input ports suffers from the state explosion problem. A possible remedy that is worth exploring in the future is to analyse each critical signal path to decide if it needs to be modelled with a more refined approximation of a signal path behaviour. For example, let us say that a critical signal path, which changes its output in a particular combination of input values only, is modelled using the one-step destabilizer method. In such a case, the output of modelled signal path changes every time the input changes, which is an unnecessary over-approximation and such a path may be modelled by using a more refined method.

Another possible research direction aims at the state explosion problem of every proposed extension method. Most of asynchronous hardware designs use some kind of the synchronization method that has already been verified. An analysis of a correct synchronizer is unnecessary if the synchronizer is correctly connected to the design (i.e., the use of the synchronizer conforms to its specification). An interesting idea is to combine the verification approaches proposed here, based on extension of the models, with using a library of synchronizers and with some light-weight static checks of their correct use. Namely, a design would be extended with a model of the transient behaviour only if the static checks were unable to verify the CDC of a circuit.

5.2 Verifying Parametrized Systems

To deal with parametrized systems, we have presented a new, quite general and automated, approach to formally verify parametrized VHDL components which was published in [93]. The approach is based on an automated translation of hardware components to counter automata and on exploiting constantly improving technology for verifying counter automata (or integer programs).¹ We have implemented a translation scheme presented in the thesis in the prototype tool *VHD2CA* [91]. It was successfully used together with the ARMC tool [83] for verification of several interesting properties of parametrized VHDL components, including a real-life component developed within the Liberouter project [80].

¹The models derived within this thesis are even being used as motivating case studies within some works aimed at improving the counter-automata analysis methods [85].

5.2.1 Future Research of Parametrized Systems

In the future, an effort can be put to lifting some of the restrictions of our initial approach, e.g., allowing a bit-wise approach to parametrized components by an automated abstraction of bit-wise operations (more details are described in Appendix C). Another interesting research direction is to investigate possibilities of reducing the size of the automata that we generate. The size of the automata can be reduced by eliminating unfeasible transitions (an unfeasible transition may be distinguished by an SMT tool), and by eliminating unreachable or irrelevant state locations. The algorithm of generating the transition relation described in Section 4.5.2, step 2b, may produce an unfeasible transition $q_1 \xrightarrow{\varphi} q_2$ for a given choice of states q_1 and q_2 since the label φ obtained by the conjunction of partially evaluated guards of behavioural rules may be unrealisable. Since some transitions may be unfeasible, it is possible that some control locations are unfeasible too and may be safely removed from generated counter automaton. The size of generated counter automaton can also be reduced by disallowing a given subset of transitions or state locations in a similar way as specifying a fairness property, i.e., by specifying which executions of a system are valid, thus other executions may be removed from the analysis. Further, we would like to do more experiments with real-life components using more different tools for handling counter automata (or integer programs), perhaps even contributing to their development by thinking of heuristics suitable for counter automata derived from hardware components.

5.3 Other Work of the Author

Apart from the work presented in the thesis, the author was engaged in two other areas applying formal verification in hardware. In both of them, the author was motivated by the work on the Liberouter [80] project. We will briefly describe the contribution these works in the next sections.

5.3.1 A Methodology of High-level Modelling and Analysis of Hardware Designs

In [94, 95], we proposed a methodology of modelling and analysis of a hardware design with the aim to perform a timing analysis of a hardware design *by using model checking*. Model checking proves if a given model meets its specification, but is unable to directly answer to questions like “What is the minimum throughput of a system?” or “What is the maximum number of items in a buffer?”.

In particular, the work proposes a methodology how to *manually* create a high-level model from a specification of a given hardware design and how to analyse it to get answers to questions similar to the above presented ones. The methodology is described on an example of the Look-up processor used within the Liberouter

project. The model checkers which we chose for the timing analysis were Uppaal² and TReX³.

Since the high-level specification of a hardware design is component-based, the modelling methodology proposed in [94, 95] also deals with components of a system. It propose suitable ways of modelling three different types of components: (i) components of buffers or communication channels, (ii) the so-called executive components dealing with processing or calculations, and (iii) environment components generating requests and examining answers. The paper provides patterns of *timed automata* [89] for every of these types of components.

Considering the timing analysis, a verification methodology based on an iterative process of model checking is presented. For instance, when dealing with the minimum throughput of a system, the work proposes to repeatedly perform model checking with increased or decreased value of a timing parameter to find a proper setting when a given system is able to process all incoming requests.

5.3.2 Proving CRC Algorithm Properties

Another work of the author aims at using formal verification to prove some interesting properties of an implementation of the cyclic redundancy check (CRC) algorithm which is commonly used in digital networks to detect incorrectly transmitted data. In particular, in [96], we presented a method how to use model checking for validating a reliability of a given CRC *generator polynomial*. A generator polynomial is a bit-vector that controls the way how a CRC hash is generated from an input message. Since two different messages may share the same hash (we say that they collide), the reliability of a generator polynomial may be represented as the minimum Hamming distance of any two input messages.

The work is based on the fact that the CRC algorithm is non-secure, i.e., there is a quite simple method how to create a colliding message to a given input message. The method is based on generating so-called *error vectors* which represent in which bits two colliding messages differ.

The problem of finding the minimum Hamming distance for a given generator polynomial is then transformed to the problem of searching an error vector with the minimal number of ones. This was performed by repeating model checking of a system which generates error vectors. The property being verified represents if there exist an error vector with less ones than a number given by the user. If the property is satisfied, one must decrease the given number and perform model checking again. The minimum Hamming distance of a generator polynomial is defined as the least number for which the property is not satisfied. The method was demonstrated on several CRC generator polynomials of the length of 16, 32, and 64 bits including CRC-ITU-T, CRC-IEEE, and CRC-ECMA.

²www.uppaal.com

³www.liafa.jussieu.fr/~sighirea/trex/

Bibliography

- [1] R. Drechsler et al. *Advanced Formal Verification*. Kluwer Academic Publishers, Dordrecht, Netherlands, 2004. ISBN: 1-4020-7721-1.
- [2] D. L. Perry, H. D. Foster. *Applied Formal Verification*. McGraw-Hill Professional. New York, USA. 2005. ISBN: 0-07-144372-X.
- [3] B. Bérard et al. *System and Software Verification. Model-Checking Techniques and Tools*. Springer Berlin / Heidelberg, 2001. ISBN: 3-540-41523-8.
- [4] K. Schneider. *Verification of Reactive Systems*. Springer Berlin / Heidelberg, 2004. ISBN: 3-540-00296-0.
- [5] D. Anastasakis, R. Damiano, H. T. Ma, and T. Stanion. A Practical and Efficient Method for Compare-Point Matching. In *Proceedings of DAC'02*, pp. 305-310. ISBN: 1-58113-461-4.
- [6] R. Drechsler. Towards Formal Verification on the System Level. In *Proceedings of RSP'04*, pp. 2–5, IEEE Computer Society, 2004. ISBN: 0-7695-2159-2.
- [7] K. McMillan. *Symbolic Model Checking*. 1993. ISBN: 0-7923-9380-5. Available online on 13 May 2010, <<http://www.kenmcmil.com/thesis.html>>.
- [8] R. E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C35(8):677-691, 1986.
- [9] R. Drechsler, D. Sieling. Binary Decision Diagrams in Theory and Practice. *International Journal on Software Tools for Technology Transfer (STTT)*. Springer Berlin / Heidelberg, 2001. ISSN: 1433-2787.
- [10] R. Bruttomesso, A. Cimatti, A. Franzen, A. Griggio, Z. Hanna, A. Nadel, A. Palti, and R. Sebastiani. A Lazy and Layered SMT(BV) Solver for Hard Industrial Verification Problems. In *Proceedings of CAV'07*, LNCS 4590, pp. 547–560, Springer Berlin / Heidelberg, 2007.

- [11] M. Aagaard, R. Jones, R. Kaivola, K. Kohatsu, C.J. Seger. Formal Verification of Iterative Algorithms in Microprocessors. In *Proceedings of DAC'00*. ACM, New York, USA, 2000.
- [12] R. Kaivola, M. Aagaard. Divider Circuit Verification with Model Checking and Theorem Proving. In *Proceedings of TPHOLs'00*, LNCS 1869, pp. 338–355, Springer Berlin / Heidelberg, 2000.
- [13] J. Harrison. Theorem Proving for Verification. In *Proceedings of CAV'08*, LNCS 5123, pp. 11–18, Springer Berlin / Heidelberg, 2008.
- [14] E.M. Clarke, O. Grumberg, and D.A. Peled. Model Checking MIT Press, 1999. ISBN: 0-262-0327-8.
- [15] IEEE Std 1850-2005. IEEE Standard for Property Specification Language (PSL). Institute of Electrical and Electronics Engineers, NJ, USA. 2005.
- [16] Z. Yang, C. Chung, I. Moon. FormalCheck Query Language Compared with CTL. 1999. Viewed in Jun 2010, <http://vlsi.colorado.edu/personal/mooni/papers/cav99_2.ps.gz>.
- [17] R. P. Kurshan. Evolution of Model Checking into the EDA Industry. In *Proceedings of ATVA'04*. LNCS 3299. pp. 2–6, Springer Berlin / Heidelberg, 2004.
- [18] A. Biere, A. Cimatti, E. M. Clarke, Y. Zhu. Symbolic Model Checking without BDDs. In *Proceedings of TACAS'99*. LNCS 1579, pp. 193–207, Springer Berlin / Heidelberg, 1999.
- [19] Z. S. Andraus, M. H. Liffiton, K. A. Sakallah. CEGAR-Based Formal Hardware Verification: A Case Study. *Technical Report CSE-TR-531-07*, University of Michigan, 2007.
- [20] H. Jain, D. Kroening, N. Sharygina, E. Clarke. Word Level Predicate Abstraction and Refinement for Verifying RTL Verilog. In *Proceedings of DAC'05*, pp. 445–450, Anaheim, California, 2005.
- [21] M. Bourahla and M. Benmohamed. Predicate Abstraction and Refinement for Model Checking VHDL State Machines. *Electronic Notes in Theoretical Computer Science*, Volume 66. Elsevier Science B. V., 2002.
- [22] S. Graf and H. Saidi. Construction of Abstract State Graphs in PVS. In *Proceedings of CAV'97*. LNCS 1254, pp. 72–83, Springer Berlin / Heidelberg, 1997. ISBN: 3-540-63166-6.
- [23] C. Flanagan and S. Qadeer. Predicate Abstraction for Software Verification. In *Proceedings of POPL'02*. ACM New York, NY, USA, 2002. ISBN: 1-58113-450-9.

- [24] T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic Predicate Abstraction of C Programs. In *ACM SIGPLAN Notices*. ACM New York, NY, USA, 2001. ISSN: 0362-1340.
- [25] E. Clarke, D. Kroening, N. Sharygina, and K. Yorav. Predicate Abstraction of ANSI-C Programs Using SAT. *Formal Methods in System Design*, Volume 25. Springer Netherlands, 2004. ISSN 1272-8102.
- [26] T. Ball, B. Cook, S. Das, and S. K. Rajamani. Refining Approximations in Software Predicate Abstraction. In *Proceedings of TACAS'04*. LNCS 2988, pp. 388–403, Springer Berlin / Heidelberg, 2004.
- [27] P. Manolios, S. K. Srinivasan, D. Vroon. Automatic Memory Reductions for RTL Model Verification. In *Proceedings of ICCAD'06*, pp. 786–793, San Jose, California, 2006.
- [28] B. Cohen. Real Chip Design and Verification Using Verilog and VHDL. Vhdl-cohen Pub, 2001. ISBN: 0-970-53942-8.
- [29] E. Tronci. Automatic Synthesis of Controllers from Formal Specifications. In *Proceedings of ICFEM'98*, p. 134, IEEE Computer Society, 1998. ISBN: 0-8186-9198-0.
- [30] K. Avnit, A. Sowmya, and J. Peddersen. ACS: Automatic Converter Synthesis for SoC Bus Protocol. In *Proceedings of TACAS 2010*, LNCS 6015, pp. 343–348, Springer Berlin / Heilderberg, 2010.
- [31] A. Pnueli. The Temporal Logic of Programs. *Proceedings of the 18th Annual Symposium on Foundation of Coputer Science*, pp. 46–57. IEEE Computer Society, 1977. ISSN: 0272-5428.
- [32] D. J. Kinniment, A. Bystrov, A. Yakovlev. Synchronization Circuit Performance. In *IEEE Journal of Solid-State Circuits*, Volume 37, pp. 202–209, 2002.
- [33] T. Chelcea, S. M. Nowick. Robust Interfaces for Mixed-timing Systems with Application to Latency-insensitive Protocols. In *Proceedings of DAC'01*, pp. 21–26., ACM New York, NY, USA, 2001. ISBN: 1-58113-297-2.
- [34] R. Ginosar. Fourteen Ways to Fool Your Synchronizer. In *Proceedings of ASYNC'03*, IEEE Computer Society, 2003.
- [35] T. Kapschitz, R. Ginosar, and R. Newton. Verifying Synchronization in Multi-Clock Domain SoC. In *Proceedings of DVCon'04*, @HDL, Inc., 2004.
- [36] T. Kapschitz and R. Ginosar. Formal Verification of Synchronizers. In *Proceedings of CHARME'05*, LNCS 3725, pp. 359–362, Springer Berlin / Heidelberg, 2005.

- [37] U. Frank, T. Kapschitz, and R. Ginosar. A Predictive Synchronizer for Periodic Clock Domains. In *Formal Methods in System Design*, Volume 28, pp. 171–186, Springer Netherlands, 2006. ISSN: 1572-8102.
- [38] B. Li and C. K. Kwok. Automatic Formal Verification of Clock Domain Crossing Signals. In *Proceedings of ASP-DAC'09*. IEEE Computer Society Press, Piscataway, NJ, USA, 2009. ISBN: 978-1-4244-2748-2.
- [39] Clock Domain Crossing – Closing the Loop on Clock Domain Functional Implementation Problems. *Technical Report*, Cadence Design System, 2004.
- [40] XILINX. Virtex-5 FPGA Data Sheet: DC and Switching Characteristics. *Product Specification*, XILINX. Viewed on 20 May 2010, <http://www.xilinx.com/support/documentation/data_sheets/ds202.pdf>.
- [41] E. Hörbs, C. Müller-Schloer, and H. Schwärtzel. Design of VLSI Circuits. Springer-Verlag Berlin, 1987. ISBN: 3-540-17663-2.
- [42] N.P. Jouppi. Timing Analysis and Performance Improvement of MOS VLSI Designs. In *IEEE Transactions on Computed-Aided Design*, Volume CAD-4, p. 650-665, 1987.
- [43] S. Devadas, K. Keutzer, and S. Malik. Delay Computation in Combinational Logic Circuits: Theory and Algorithms. In *Proceedings of ICCAD 1991*, pp. 176–179, IEEE Computer Society, 1991.
- [44] A.I. Kayssi, K.A. Sakallah, and T.N. Mudge. The Impact of Signal Transition Time on Path Delay Computation. In *Circuits and Systems II: Analog and Digital Signal Processing*. IEEE Transactions, Volume 40, Issue 5, 1993.
- [45] Mentor Graphics. *0-In Formal Verification Data Sheet*, 2006. Viewed in May 2010, available from <http://www.mentor.com/products/fv/0-in_fv/>.
- [46] Axiom Design Automation. *@HDL Verifier*, 2004. Viewed on 7 June 2010, <<http://www.athdl.com/Verifier>>.
- [47] T. F. Melham. Formalizing Abstraction Mechanisms for Hardware Verification in Higher Order Logic. *Technical Report no. 201*, University of Cambridge, 1990.
- [48] E. Clark, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-Guided Abstraction Refinement. In *Proceedings of CAV'00*, LNCS 1855, pp. 154–169, Springer Berlin / Heidelberg, 2000.
- [49] J. O'Leary, X. Zhao, R. Gerth, and C-J. H. Seger. Formally Verifying IEEE Compliance of Floating-Point Hardware. *Intel Technology Journal*, Q1, 1999.

- [50] D. Russinoff. A Mechanically Checked Proof of IEEE Compliance of the Floating-Point Multiplication, Division, and Square Root Algorithms of the AMD-K7 Processor. In *London Mathematical Society Journal of Computation and Mathematics*, 1998.
- [51] D. Monniaux. The Pitfalls of Verifying Floating-point Computations. In *Proceedings of TOPLAS'08*, Volume 30, Issue 3. ACM New York, NY, USA, 2008. ISSN: 0164-0926.
- [52] C. Pasarean, W. Visser. Verification of Java Programs Using Symbolic Execution and Invariant Generation. In *Proceedings of SPIN'04*, LNCS 2989, Springer Berlin / Heidelberg, 2004.
- [53] J. Kořenek, T. Pečenka, and M. Žádník. NetFlow Probe Intended for High-Speed Networks. In *Proceedings of FPL'05*. IEEE Computer Society, 2005.
- [54] T. Ly, N. Hand, and Ch. Ka kei Kwok. Formally Verifying Clock Domain Crossing Jitter Using Assertion-Based Verification. In *Proceedings of DVCon'04*. Verilab, Munich, DE, 2004.
- [55] M. Litterick. Pragmatic Simulation-Based Verification of Clock Domain Crossing Signals and Jitter using SystemVerilog Assertings. In *Proceedings of DVCon'06*. Verilab, Munich, DE, 2006.
- [56] K.L. McMillan. Cadence SMV. Viewed in May 2010, <<http://www.kenmcmil.com/smv.html>>.
- [57] K.L. McMillan. Getting Started with SMV. Tutorial to SMV, Cadence Berkeley Labs, CA, USA. 1999. Viewed in Jul 2010, <<http://www.kenmcmil.com/psdoc.html>>.
- [58] J. F. Wakerly. Digital Design: Principles and Practices. 4th edition. Prentice-Hall, India, 2005. ISBN: 8-120-33021-8.
- [59] B. Lin and S. Devadas. Synthesis of Hazard-free Multi-level Logic Under Multiple-input Changes from Binary Decision Diagrams. In *Proceedings of ICAAD'94*. IEEE Computer Society Press, Los Alamitos, CA, USA. 1994. ISBN: 0-89791-690-5.
- [60] H. Comon, Y. Jurski. Multiple Counters Automata, Safety Analysis and Presburger Arithmetic. In *Proceedings of CAV'98*, LNCS 1427, Springer Berlin / Heidelberg, 1998.
- [61] A. Bouajjani, M. Bozga, P. Habermehl, R. Iosif, P. Moro, and T. Vojnar. Programs with Lists are Counter Automata. In *Proceedings of CAV'06*, LNCS 4144, Springer Berlin / Heidelberg, 2006.

- [62] S. Bardin, A. Finkel, J. Leroux, and L. Petrucci. FAST: Fast Acceleration of Symbolic Transition systems. In *Proceedings of CAV'03*, LNCS 2725, Springer Berlin / Heidelberg, 2003.
- [63] S. Bardin, A. Finkel, and E. Lozes. From Pointer Systems to Counter Systems Using Shape Analysis. In *Proceedings of AVIS'06*, 2006.
- [64] S. Chaki, E. Clarke, A. Groce, J. Ouaknine, O. Strichman, and K. Yorav. Efficient Verification of Sequential and Concurrent C Programs. *Formal Methods in System Design*, 25(2–3):129–166, 2004.
- [65] P.P. Chu. *RTL Hardware Design Using VHDL: Coding for Efficiency, Portability, and Scalability*. John Wiley and Sons, Inc., Hoboken, New Jersey, 2006.
- [66] P. Habermehl, R. Iosif, A. Rogalewicz, and T. Vojnar. Proving Termination of Tree Manipulating Programs. In *Proceedings of ATVA'07*, LNCS 4762, 2007. Springer Berlin / Heidelberg, 2007.
- [67] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software Verification with Blast. In *Proceedings of SPIN'03*, LNCS 2648, Springer Berlin / Heidelberg, 2003.
- [68] T. Yavuz-Kahveci, C. Bartzis, and T. Bultan. Action Language Verifier, Extended. In *Proceedings of CAV'05*, LNCS 3576, Springer Berlin / Heidelberg, 2005.
- [69] M. L. Minsky. *Computation: Finite and Infinite Machines* Prentice-Hall International, 1967. ISBN: 0-13-165563-9
- [70] IEEE Std 754-1990. Revision of IEEE Std 754, 1985 Edition. IEEE Standard for Binary Floating-Point Arithmetic. Institute of Electrical and Electronics Engineers, NJ, USA. 1990.
- [71] IEEE Std 854-1994. Revision of IEEE Std 854, 1987 Edition. IEEE Standard for Radix-Independent Floating-Point Arithmetic. Institute of Electrical and Electronics Engineers, NJ, USA. 1994. ISBN: 0-7381-1167-8.
- [72] IEEE Std 1129.1-1998. IEEE Standard for Waveform and Vector Exchange (WAVES). Institute of Electrical and Electronics Engineers, NJ, USA. 1998. ISBN: 0-7381-3032-X.
- [73] IEEE Std 1076-2002. IEEE Standard VHDL Language Reference Manual. Revision of IEEE Std 1076, 2000 Edition. Institute of Electrical and Electronics Engineers, NJ, USA. 2002. Pages 300. 2000. ISBN: 0-7381-3247-0.

- [74] IEEE Std 1076.1-2007. IEEE Standard VHDL Analog and Mixed-Signal Extensions. Institute of Electrical and Electronics Engineers, NJ, USA. 2007. ISBN: 0-7381-5627-2.
- [75] IEEE Std 1076.2-1996. IEEE Standard VHDL Mathematical Packages. Institute of Electrical and Electronics Engineers, NJ, USA. 1996. ISBN: 1-55937-894-8.
- [76] IEEE Std 1076.3-1997. IEEE Standard VHDL Synthesis Packages. Institute of Electrical and Electronics Engineers, NJ, USA. 1997. ISBN: 1-55937-923-5.
- [77] IEEE Std 61691-5-2004. Behavioural Languages - Part 5: VITAL ASIC (Application Specific Integrated Circuit) Modeling Specification. Institute of Electrical and Electronics Engineers, NJ, USA. 2004. ISBN: 2-8318-7684-2.
- [78] IEEE Std 1076.6-1999. IEEE Standard for VHDL Register-Transfer Level Synthesis. Institute of Electrical and Electronics Engineers, NJ, USA. 1999. ISBN: 0-7381-1819-2.
- [79] IEEE Std 1164-1993. IEEE Standard Multivalued Logic System for VHDL Model Interoperability (Std_logic_1164). Institute of Electrical and Electronics Engineers, NJ, USA. 1993. ISBN: 0-7381-0991-6.
- [80] Libero Project Homepage. Viewed on Oct 2009, <<http://www.liberouter.org/>>.
- [81] Mentor Graphics. *Leonardo Spectrum Data Sheet*, 2008. Viewed in May 2010, available from <http://www.mentor.com/products/fpga_pld/synthesis/leonardo_spectrum>.
- [82] M. Litterick. Using SystemVerilog Assertions for Functional Coverage. In *Proceedings of DAC'05*. Verilab, Munich, DE, 2005.
- [83] A. Podelski, A. Rybalchenko, ARMC: The Logical Choice for Software Model Checking with Abstraction Refinement. In *Proceedings of PADL'07*, LNCS 4354, Springer Berlin / Heidelberg, 2007.
- [84] M. Bozga, P. Habermehl, R. Iosif, F. Konečný, and T. Vojnar. Automatic Verification of Integer Array Programs. In *Proceedings of CAV'09*, LNCS 5643, pp. 157–172, Springer Berlin / Heidelberg, 2009.
- [85] R. Iosif, M. Bozga, and F. Konečný. Fast Acceleration of Ultimately Periodic Relations. To appear in *Proceedings of Computer Aided Verification 2010*.
- [86] S. Magill, M.-H. Tsai, P. Lee, Y.-K. Tsay. Automatic Numeric Abstractions for Heap-Manipulating Programs. In *Proceedings of POPL'2010*, ACM SIGPLAN Notices, Volume 45, 2010. ISSN: 0362-1340.

- [87] S. Bardin, J. Leroux, G. Point. FAST Extended Release. In *Proceedings of CAV'06*, LNCS 4144, Springer Berlin / Heidelberg, 2006.
- [88] C. Alias, A. Darte, P. Feautrier, and L. Gonnord. Multi-dimensional Rankings, Program Termination, and Complexity Bounds of Flowchart Programs. Accepted in *Static Analysis Symposium*, 2010.
- [89] R. Alur. Timed Automata. In *Proceedings of CAV'99*, LNCS 1633, pp. 8–22, Springer Berlin / Heidelberg, 1999.
- [90] A. Smrčka. CDCreveal: A Tool for Extending Zero-delay Models of Asynchronous Circuits. *Project's homepage*. Revised on 25 July 2010, <<http://www.fit.vutbr.cz/~smrcka/tools/cdcreveal/>>.
- [91] A. Smrčka. VHD2CA: A Translator From VHDL to Counter Automata. *Project's homepage*. Revised on 1 Aug. 2010, <<http://www.fit.vutbr.cz/~smrcka/tools/vhd2ca/>>.
- [92] A. Smrčka et al. Verifying VHDL Design with Multiple Clocks in SMV. In *Proceedings of FMICS'06*, LNCS 4346, pp. 148–164, Springer Berlin / Heidelberg, 2007.
- [93] A. Smrčka and T. Vojnar. Verifying Parametrised Hardware Designs Via Counter Automata. In *Proceedings of HVC'07*, LNCS 4899, pp. 51–68, Springer Berlin / Heidelberg, 2008.
- [94] P. Matoušek, A. Smrčka, and T. Vojnar. High-level Modelling, Analysis, and Verification on FPGA-based Hardware Designs. In *Proceedings of CHARME'05*, LNCS 3725, pp. 371–375. Springer Berlin / Heidelberg, 2005.
- [95] P. Matoušek, A. Smrčka, and T. Vojnar. High-level Modelling, Analysis, and Verification on FPGA-based Hardware Designs. CESNET Technical Report no. 8/2005. Viewed in July 2010, <<http://www.cesnet.cz/doc/techzpravy/2005/lup/lup.pdf>>.
- [96] A. Smrčka et al. Formal Verification of the CRC Algorithm Properties. In *Proceedings of MEMICS'06*, pp. 55–65, Mikulov CZ, 2006. ISBN: 80-214-3287-X.

Appendix A

An Example of the Implementation of the Destabilizer

At this appendix, we will show an example of application of extension of critical signal paths in Cadence SMV. For the reference, we will use the design which structural diagram is depicted in Figure A.1.

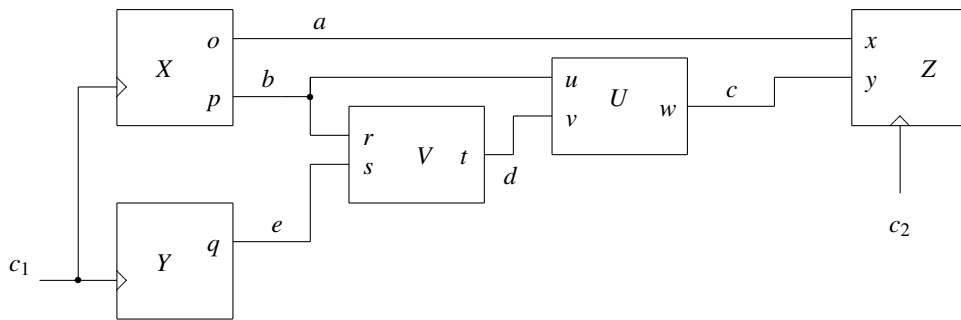


Figure A.1: An example of hardware design with multiple clocks.

The zero-delayed model in Cadence SMV of a hardware design described above would be as depicted in Figure A.2. The hardware design has two clock signals c_1 and c_2 , four critical signal paths:

$$\rho_1 = XoaxZ$$

$$\rho_2 = XpbuUwcyZ$$

$$\rho_3 = XpbrVtdvUwcyZ$$

$$\rho_4 = YqesVtdvUwcyZ$$

```

1  -- module definitions
2  module X(clk, o, p) {...}
3  module Y(clk, q) {...}
4  module V(r, s, t) {...}
5  module U(u, v, w) {...}
6  module Z(clk, x, y) {...}
7
8  -- signal declarations
9  c1, c2: boolean;
10 b, c, d, e: boolean;
11
12 -- module instantiations
13 g1: X(c1, a, b);
14 g2: Y(c1, e);
15 g3: V(b, e, d);
16 g4: U(b, d, c);
17 g5: Z(c2, a, c);

```

Figure A.2: An example of the zero-delayed model. Note that $\{\dots\}$ in a module definition represents a behaviour of a given module.

and two sets of signal paths with the same destination port:

$$[\rho_1] = \{\rho_1\}$$

$$[\rho_2] = [\rho_3] = [\rho_4] = \{\rho_2, \rho_3, \rho_4\}$$

That is, we need to create two destabilizers δ_{ρ_1} and δ_{ρ_2} sensitive for one, i.e., $\{\sigma_i(\rho_1)\} = \{a\}$, and two, i.e., $\{\sigma_i(\rho_2), \sigma_i(\rho_3), \sigma_i(\rho_4)\} = \{b, b, e\} = \{b, e\}$, input signals change. The first destabilizer produces the x-value for the length of 1 verification step, while the latter one destabilizes its output for 3 steps:

$$\max(\{|i(\rho_2)|, |i(\rho_3)|, |i(\rho_4)|\}) = \max(\{2, 3, 3\}) = 3.$$

Also, two new signals s_1 and s_2 must be declared to connect destabilizers to given input ports. The transformed version of the zero-delay model is shown below:

```

1  -- module definitions
2  module X(clk, o, p) {...}
3  module Y(clk, q) {...}
4  module U(u, v, w) {...}
5  module V(r, s, t) {...}
6  module U(u, v, w) {...}
7  module Z(clk, x, y) {...}
8  module Destabilizer1(i1, alpha, omega) {
9      input i1 : boolean;
10     input alpha : boolean;
11     output omega : boolean;
12     cnt : 0..1;
13     init(cnt) := 0;
14     next(cnt) := case {
15         i1!=next(i1) : 1;
16         cnt=1       : 0;
17         cnt>0 & cnt<1 : cnt+1;
18         1           : 0;
19     }
20     omega := case {
21         cnt=0 : alpha;
22         cnt>0 : {0,1};
23     }
24 }
25 module Destabilizer2(i1, i2, alpha, omega) {
26     input i1 : boolean;
27     input i2 : boolean;
28     input alpha : boolean;
29     output omega : boolean;
30     cnt : 0..3;
31     init(cnt) := 0;
32     next(cnt) := case {
33         i1!=next(i1) | i2!=next(i2) : 1;
34         cnt=3                       : 0;
35         cnt>0 & cnt<3               : cnt+1;
36         1                           : 0;
37     }
38     omega := case {
39         cnt=0 : alpha;
40         cnt>0 : {0,1};
41     }
42 }

```

```
44 -- signal declarations
45 c1, c2: boolean;
46 a, b, c, d, e: boolean;
47 s1, s2: boolean;
48
49 -- module instantiations
50 g1: X(c1, a, b);
51 g2: Y(c1, e);
52 g3: V(b, e, d);
53 g4: U(b, d, c);
54 d1: Destabilizer1(a, a, s1);
55 d2: Destabilizer2(b, e, c, s2);
56 g5: Z(c2, s1, s2);
```

Appendix B

Simple Handshake Synchronization Protocol

Here is the source code of a handshake data transfer synchronization protocol. The core of the synchronization is inspired from four-phase handshake, widely used in many transactional protocols. The model consists of three modules—module `main` which defines an environment of the module `transmitter`, the control part of the synchronization, and module `fdce` which implements a generic flip-flop gate. The metastability is dealt with two-flip-flops solution, e.g., lines 47-54. The assertion at line 15 specifies the consistent data (for the same information of two bits on the transmitter input, the output must be also the same at any time). An example of flaw in the design is presented at line 68, where the definition of `rdy` signal (informing the receiver that new data is ready) does not depend on the receiver acknowledgement.

```
1 module main()
2 {
3     a: boolean; -- data sent
4     b: boolean; -- the first part of received data
5     c: boolean; -- the second part of received data
6     clk1: boolean;
7     clk2: boolean;
8
9     a := {0,1};
10    clk1 := {0,1};
11    clk2 := {0,1};
12
13    t1: transmitter(a, a, clk1, b, c, clk2);
14
15    data_consistent: assert G(b = c);
16 }
```

```

17 module transmitter(din_0, din_1, clk_wr, dout_0, dout_1, clk_rd)
18 {
19     input din_0: boolean;
20     input din_1: boolean;
21     input clk_wr: boolean;
22     input clk_rd: boolean;
23     output dout_0: boolean;
24     output dout_1: boolean;
25
26     dlink_0: boolean;
27     dlink_1: boolean;
28     rdy: boolean;
29     ack: boolean;
30
31     -- synchronize cdc signals (metastability) via 2FF
32     c_ack: boolean;
33     s1_s: boolean;
34
35     c_rdy: boolean;
36     s2_s: boolean;
37
38     c_dlink_0: boolean;
39     s3_s: boolean;
40
41     c_dlink_1: boolean;
42     s4_s: boolean;
43
44     pwr: boolean;
45
46     pwr := 1;
47     s1_f1: fdce(s1_s, ack, clk_wr, pwr);
48     s1_f2: fdce(c_ack, s1_s, clk_wr, pwr);
49     s2_f1: fdce(s2_s, rdy, clk_rd, pwr);
50     s2_f2: fdce(c_rdy, s2_s, clk_rd, pwr);
51     s3_f1: fdce(s3_s, dlink_0, clk_rd, pwr);
52     s3_f2: fdce(c_dlink_0, s3_s, clk_rd, pwr);
53     s4_f1: fdce(s4_s, dlink_1, clk_rd, pwr);
54     s4_f2: fdce(c_dlink_1, s4_s, clk_rd, pwr);
55
56     -- implement both parts of transmitter
57     lrdy: boolean;
58     s_ce: boolean;
59     rdy_val: boolean;
60     nack: boolean;
61     nack := ~c_ack;
62     s_ce := ~lrdy & nack;
63     reg_lrdy: fdce(lrdy, nack, clk_wr, pwr);

```

```

64      /***** GOOD *****/
65      rdy_val := (~rdy & lrdy) | (rdy & nack);
66      reg_rdy: fdce(rdy, rdy_val, clk_wr, pwr);
67      /***** BUG *****/
68      /*rdy := lrdy;*/
69
70      tr1_s0: fdce(dlink_0, din_0, clk_wr, s_ce);
71      tr1_s1: fdce(dlink_1, din_1, clk_wr, s_ce);
72      reg_ack: fdce(ack, c_rdy, clk_rd, pwr);
73      r0: fdce(dout_0, c_dlink_0, clk_rd, c_rdy);
74      r1: fdce(dout_1, c_dlink_1, clk_rd, c_rdy);
75  }
76
77  module fdce(q, d, c, ce)
78  {
79      output q: boolean;
80      input d: boolean;
81      input c: boolean;
82      input ce: boolean;
83
84      init(q) := 0;
85
86      if (ce & ~c & next(c))
87          next(q) := d;
88      else
89          next(q) := q;
90  }

```

Appendix C

Examples of Abstractions of Bit-wise Operations in VHDL

In the following, some of ideas of an abstraction of a bit-wise operation is introduced. We will present the ideas on the example of a simple arithmetic unit in Figure C.1.

The first idea is to reduce the number of one-bit variables by representing multiple one-bit variables as a counter. The idea aims at the bit-vectors that are accessed via a bit-vector literals only (cf. Figure C.1, lines 27–32). The proposed method of transforming the model to the counter automata deals with such a bit-vector as with two independent one-bit variables. As a result, two more bit variables increase the size of a counter automaton (in a number of state locations) four times larger. When representing the bit-vector as a counter, the size of an automaton is dramatically reduced, but the number transitions slightly rises (the access to a bit-vector/counter value must be represented by an arithmetic expression). In particular, the bit-vector `opcode` can be addressed using a counter if it is accessed by a comparison with an integer number (cf. Figure C.2).

Another idea aims at a single-bit access of a bit-vector (cf. Figure C.1, line 35). In particular, we replace a bit-wise access with arithmetic operations over a counter value. For a example, in Figure C.1, the abstraction of the bit-wise access can be replaced with the statements depicted in Figure C.3. Note that such an abstraction is not easy to obtain and needs a great deal of investigation to the roots of the problem.

```

1  library ieee;
2    use ieee.std_logic_1164.all;
3    use ieee.std_logic_arith.all;
4  entity arith_unit is
5    generic ( WIDTH : integer = 64 );
6    port(
7      clk : in std_logic;
8      reset : in std_logic;
9      opcode: in std_logic_vector(0 to 1);
10     a, b: in signed(0 to WIDTH-1);
11     c: out signed(0 to WIDTH-1);
12     signed : out std_logic
13   );
14 end entity arith_unit;
15 architecture my_unit of arith_unit is
16   signal a1, a2, n1, n2: signed(0 to WIDTH-1);
17 begin
18   a1 <= a + b;
19   a2 <= a - b;
20   n1 <= -a;
21   n2 <= -b;
22   process (reset, clk)
23   begin
24     if (reset = '1')
25       c <= (others => '0');
26     elsif (clk'event and clk = '1') then
27       case opcode is
28         when '00' => c <= a1;
29         when '01' => c <= a2;
30         when '10' => c <= n1;
31         when '11' => c <= n2;
32       end case;
33     end if;
34   end process;
35   signed <= c(WIDTH-1);
36 end architecture;

```

Figure C.1: A simple arithmetic unit. Note the unit is defined by a generic entity (cf. parameter WIDTH) and contains bit-wise operations (bit-wise case statement at lines 27–32 and a bit-wise access to a register value at line 35).

```

27     case opcode is
28         when 0 => c <= a1;
29         when 1 => c <= a2;
30         when 2 => c <= n1;
31         when 3 => c <= n2;
32     end case;

```

Figure C.2: All literals of bit-vectors are substituted with appropriate positive integer numbers.

```

35     case opcode is
36         when 0 => signed <= (a1 < 0) or (a1 >= 2**WIDTH);
37         when 1 => signed <= ((a2 < 0) and (a2 >= -2**WIDTH))
38                             or (a2 >= 2**WIDTH);
39         when 2 => signed <= (n1<0);
40         when 3 => signed <= (n2<0);
41     end case;

```

Figure C.3: A simple bit-wise access $\text{signed} \leq c(\text{WIDTH}-1)$ replaced with rather complicated statement containing only the reference to a counter values. Note that the analysis of the left-most bit of a bit-vector c should take into account also the possible overflow of the arithmetic operations over limited bit-vectors (the expressions $2**\text{WIDTH}$).