Verification of Asynchronous and Parametrized Hardware Designs

Aleš Smrčka*
Department of Intelligent Systems
Faculty of Information Technology
Brno University of Technology
Božetěchova 2, 612 66 Brno, Czech republic
smrcka@fit.vutbr.cz

Abstract

Two original approaches to formal verification of hardware designs are introduced. 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.

Categories and Subject Descriptors

B.4.4 [**Input/Output and Data Communications**]: Performance Analysis and Design Aids; B.8.2 [**Performance and Reliability**]: Performance Analysis and Design Aids; D.2.4 [**Software Engineering**]: Software/Program Verification—*Model checking*

Keywords

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

*Recommended by thesis supervisor:
Assoc. Prof. Tomáš Vojnar
Defended at Faculty of Information Technology, Brno University of Technology on December 23, 2010.

Smrčka, A. Verification of Asynchronous and Parametrized Hardware Designs. Information Sciences and Technologies Bulletin of the ACM Slovakia, Vol. 2, No. 2 (2010) 60-69

1. Introduction

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: simulation and testing, and formal analysis and verification. The main idea of *formal verification* is to prove the correctness of a design instead of performing some random test cases. Different techniques for the proof process have been proposed during last decades, one of which is *model checking* [3]. Model checking consists in verifying that a model of a system (or sometimes the system itself) satisfies a given property by exhaustively exploring the state space. If a violation of a property 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 [3]) or in some specialized property specification language. Each method of formal verification can be classified using two criteria: soundness and completeness. A *sound method* is capable to detect all possible errors, i.e., if the positive answer guarantees the correctness of a system. A *complete method* guarantees that the counter-example found in a model is also feasible in the real system.

Since the size of state space grows exponentially with the size of a system, model checking greatly suffers from state space explosion problem. To cope with state-space explosion problem, different techniques in hardware verification have proven to be useful such as, *abstraction methods* simplifying a system before verifying it. An abstraction (modelling) of a system may be classified as an under-approximation (model checking of an under-approximated model is complete, but unsound) or an over-approximation (allowing the proof do be sound). An 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.

Even though model checking is successfully applied and has become the state-of-the-art in many design flows, still many problems remain [1]. Here, we aim at two of such problems: (i) *multiple clocks* of a circuit, or (ii) *parametrization of designs*. When verifying a multiple-clock circuit, one must take into account that the digital data in a real circuit are transferred via analog signals. Unfortunately, current modelling techniques suitable for model checking assume that the circuit is purely digital, thus errors due to delays of a real circuit may be missed. When dealing with parametrized designs (parametrization is widely used when creating libraries of re-usable hardware components), the problem appears if the number of different configurations of parametric modules is unbound. In such a case, the state space is infinite, but the current model checkers used in a hardware design flow work on a finite state-space basis.

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 get involved in verifying clock domain crossing and even to understand synchronization problems. 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 synchronization faults.

The second aim of the thesis is to facilitate formal verification of parametrized designs. Currently, verification of parametrized 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.

Structure of the paper. The following section introduces the reader to to the basics of hardware design HDL languages, and to the current techniques used for modelling such designs for their subsequent model checking. The Sections 3 and 4 discuss both of the original contributions. 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 description of the proposed techniques is provided. In particular, Section 3 deals with verification of designs with multiple clocks and provides four approaches for model checking such designs. In Section 4, we discuss the area of verification of parametrized hardware designs via transformation of an HDL design to a counter automaton. A general summary of discussed methods provides Section 5.

2. Background

2.1 Hardware Designs in RTL

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/SystemC language), hardware developers still use less abstract level of a hardware description, *register transfer level* (RTL). In RTL, 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.

Different Types of a Description. The *dataflow description* specifies the relation between ports or signals by predefined arithmetic, logic, or conditional connectors. In *structural description*, 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. The *behavioural description* 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. 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., combinational logic gates, simple registers, counters), while the structural description is used for building more complex components or the entire system.

2.2 Modelling Hardware Designs

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. However, especially when dealing with asynchronous circuits, we have to note that 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.

In this work, we use so-called *zero-delay modelling* which, in contrast to widely used *modelling of synchronous circuits* (as described in [3]), gives emphasis to events on clock signals. 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. 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.*

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, in LTL, $(\mathbf{G}\ \mathbf{F}\ clk) \wedge (\mathbf{G}\ \mathbf{F}\ \neg clk)$ 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 $(\neg clk \wedge\ \mathbf{X}\ clk)$ for a rising edge of the signal clk and $(clk \wedge\ \mathbf{X}\ \neg clk)$ for a falling edge.

3. Verifying Designs with Multiple Clocks

In this section, 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 [4]. 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 [5]. Several solutions have been proposed: the simple one-wire solution via a two-flip-flop synchronizer [6], a dual-clock FIFO synchronization channel [7], or predictive synchronizers [8] for periodic clock domains. In [9] and [10], 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 [11, 12], using simulation-based verification with System Verilog assertions [13], or using automatic formal verification [14].

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 [9], 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 Modelling Asynchronous Behaviour

In the following, 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. 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.

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 [4], 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 Basic Idea. The basic idea of all four proposed methods lies in generating a random phase to the propagation of a signal through a critical signal path whenever a signal changes. 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[1].

[1]In the following, by introducing a random value on some signal we mean the assigning a non-deterministic choice of values 0 and 1 to the signal.

In general, the proposed approach is based on two assumptions: (1) clock signals are random all the time, 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: (i) The destination clock domain reads a signal before its change, (ii) a signal is sensed within a random phase representing a rising or a falling edge of a signal, and (iii) 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.

Definitions. 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 $I \cap O = \emptyset$. And finally, $M : P \to S$ is a *signal interconnection function.*

A hardware design $H = (S, C, P, G, M)$ is *valid* iff: (1) 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.

A *signal path* π is a string of gates and their I/O ports interleaved by signals connecting the ports (…, gate, output port, signal, input port, gate, …). Formally, for a hardware design $H = (S, C, P, G, M)$, a *signal path* $\pi = \langle g_1 o_1 s_1 i_2 g_2 o_2 s_2 i_3 g_3 \dots g_{n-1} o_{n-1} s_{n-1} i_n g_n \rangle$ of length $n > 1$ is a connected sequence of gates, ports, and signals such that $\forall j \in \{1, ..., n-1\} : g_j = (c_j, I_j, O_j) \in G \wedge g_{j+1} = (c_{j+1}, I_{j+1}, O_{j+1}) \in G \wedge o_j \in O_j \wedge i_{j+1} \in I_{j+1} \wedge s_j \in S \wedge (o_j, s_j) \in M \wedge (i_{j+1}, s_j) \in M$. A *critical* signal path is a path which starts with a gate in some clock domain and ends in a gate in another clock domain. We call *critical input ports* as all input ports of a critical signal path. Further, we define *a destination port* being the input port which appears at the end of a given critical signal path. Finally, we define *a clock domain output* being a critical input port which appears at the beginning of a critical signal path.

3.2.1 Extending All Critical Input Ports

We now introduce 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.

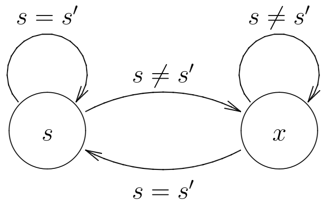


Figure 1: The automaton of a delayed input gate.

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 virtual gate. The new gate delays the original signal value before propagating it to the original input—we call the new gate a *delayed input* Δ. We introduce the behaviour of Δ by the finite automaton in Fig. 1. The arcs represent if values of preceding signal s, originally connected to the critical input port, differ in the current s and future state s', the control states identify values of output of Δ (in the state s, the delayed input holds a value of signal s, state x represents a random value).

3.2.2 Extending Critical Paths

Previously proposed method is rather precise but may cause a significant state-space explosion due to the number of newly introduced state variables. We try to avoid this explosion by introducing a less precise, approximate model. The previous approach delays a propagation of a new stable value through a critical signal path with a single step per each critical input port. We over-approximate such a behaviour by a phase of random values on the output of critical signal path such that the phase takes at least the number of steps as a cumulated delay generated by the previous method. We introduce *a destabilizer* as a virtual gate we use to generate a phase of random values on the output of a critical signal path. A destabilizer will be connected to the destination port of a critical signal path where random values can become visible. We create one destabilizer δ_ρ for every set of critical signal paths having the same destination port (we denote $[\rho]$ being the set of paths ρ with the same destination port). The destabilizer starts to generate random values if one of the clock domain outputs of considered signal paths changes its value. The length of a random phase is given by a length (a number of input ports) of the longest critical signal path of $[\rho]$.

3.2.3 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. 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 zerodelayed 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 δ_1 added to the output of a critical signal path ρ randomizes the output shared with critical signal paths $[\rho]$. The behaviour of a one-step destabilizer is the following. In a stable state, a one-step destabilizer produces a value of the original destination port 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.

3.2.4 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 clock domain outputs of the signal paths sense a value change. 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. To refine such a model, we use the logic of the combinational gates of a critical signal path to restrict the randomness. We let every clock domain output 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. If the logic is designed such that it does not propagate a change further to the system, the destination port of a given critical signal path does not sense the change.

3.3 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. 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 performed the experiments on good and buggy versions of these components (a buggy version incorrectly implements the synchronization).

We implemented all proposed modelling methods in tool *CDCreveal* which uses Cadence SMV as an input language. We successfully used the tool on both asynchronous units. Table 1 and Table 2 show the statistics of a verified model and the size of its state space: The first column identifies the used *method* (*noext* means that no modelling of CDC was performed, *dinput* represents the method of extending all critical input ports, *destabil* represents the method using plain destabilizers, *osd* stands for one-step destabilizers, and *doutput* represents the method of extending clock domain outputs), 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, and *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. 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.

method	var.	trans.	reach. set	satisfied
noext	18	194	2251	true
dinput	22	226	4255	true
destabil	28	266	8431	true
osd	22	226	4255	true
doutput	22	226	4255	true

(a) A correctly implemented handshake

method	var.	trans.	reach. set	satisfied
noext	17	180	1598	true
dinput	20	204	4016	false
destabil	25	240	7817	false
osd	21	212	4538	false
doutput	20	204	4016	false

(b) A buggy handshake protocol

Table 1: Experimental results with model checking a simple handshake synchronization protocol.

method	var.	trans.	reach. set	satisfied
noext	41	873	32224	true
dinput	179	3123	N/A	N/A
destabil	61	1269	164042	false
osd	43	953	37314	false
doutput	73	1691	65533	true

(a) Correctly implemented synchronization

method	var.	trans.	reach. set	satisfied
noext	41	845	33213	true
dinput	179	3111	N/A	N/A
destabil	61	1241	155346	false
osd	43	925	69823	false
doutput	73	1659	70134	false

(b) Wrong synchronization of counters

Table 2: Experiment results of model checking the asynchronous FIFO.

4. Verifying Parametrized Hardware Designs

In this section, 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 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 (synthesisable) 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 [15], we have been able to verify several non-trivial properties of parametrized VHDL components, including a real-life component implementing an asynchronous queue.

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., [16, 17]). 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 [18, 19, 20, 21].

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 [22], 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 instances.

4.2 Preprocessing VHDL for Translation

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 in VHDL to a uniform description using a small set of language constructions.

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 without most of IEEE library extensions, we do not allow a bit-wise access to variables with a parametric range and we do not allow `for` loops over parametrized variables, and we also disallow a use of procedures and functions.

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. 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 for all the subsequent transformation steps. The goal of the simplification is to obtain code consisting of conditional signal assignments only.

Handling Integer Variables. 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 `v1 <= v2+v3;` over integer variables represented on n bits has to be translated to an assignment of the form $v_1 := (v_2 + v_3) \ mod \ 2^n$). 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.

4.3 An Intermediate Behavioural Model

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.

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, $T : V \to \{\texttt{bool}, \texttt{int}\}$ 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 $\overline{V} = V \times \{\texttt{last}, \texttt{next}, \texttt{posedge}, \texttt{negedge}\}$ to be the set of possible references to the values of variables from V with the following meaning: $(v, \texttt{last}) \in \overline{V}$ refers to the value of v in the *last reached* (i.e., current) *state*—in expressions, we usually abbreviate it simply to v, $(v, \texttt{next}) \in \overline{V}$, abbreviated to v', denotes the value of v in the *next state*, $(v, \texttt{posedge}) \in \overline{V}$, 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 $T(v) = \texttt{bool}$), and $(v, \texttt{negedge}) \in \overline{V}$, 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 $T(v) = \texttt{bool}$). Further, let E be the set of all (well-typed) expressions that one can form over $\overline{V}$ using arithmetical $(+, -, *, ...)$, relational $(=, \neq, <, >, \leq, \geq)$, and logical $(\neg, \wedge, \vee, ...)$ 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 $c \to v := e$ 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 $c = c_1 c_2 ... c_n$ says that if $c_1 \wedge c_2 \wedge ... \wedge c_n$ holds for a given valuation of the variables, v will get a new value obtained by a valuation of e. If $c = \varepsilon$, we consider it to be always true, and the assignment $v := e$ is always enabled.

For a behavioural rule $b : c \to v := e \in B$, we denote: $cond(b) = c$ the enabling condition of b, $var(b) = v$ the variable to be set, $value(b) = e$ the expression defining the new value of v. Further let, $F(e)$, for $e \in E \cup C^*$, be the set of reference to variables occurring in e, and $B(v) = \{b \mid b \in B, var(b) = v\}$ be the set of behavioural rules over a variable v.

4.3.1 Extracting Behavioural Rules from VHDL

In order to obtain the set of behavioural rules B from a description consisting of if-then-else statements, we extract the rules from VHDL statements as follows: For each parallel assignment `v <= e;`, we add a rule $\varepsilon \to v := e$ into B. For each sequential process that sets a variable v by a single, possibly nested, `if` statement (after the preprocessing, there is no other possibility), we proceed as follows. For each assignment statement `v <= e;` that appears on the leaf level of such a (nested) `if` statement, we add a rule $c'_1, c'_2, ..., c'_n \to v := e$ into B ($n \geq 1$). Here, $c_1, c_2, ..., c_n$ are all the branching conditions that one tests before reaching `v <= 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 Fig. 2.

4.3.2 Adjustments of Behavioural Rules

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. For every such an input $v \in V_i$, we add the following behavioural rules: If v is a 1-bit variable (i.e., $T(v) = bool$), we add the rules $\varepsilon \to v := v$ and $\varepsilon \to v := \neg v$ to B. Otherwise (i.e., $T(v) = int$), we extend B by the rule $\varepsilon \to v := random$. Here, *random* represents a random integer value.

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. 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. 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. Let further $V_s = V_i \cup \{v \mid v \in V, cond(b) \neq \varepsilon\}$ be the set of state variables. Before generating counter automata, we change the intermediate behavioural model to use the state variables only.

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

Since gates working in transparent and synchronous mode differ in a behaviour, we manifest this effect in behavioural rules. A gate working in transparent mode produces output which reflects current values on the input, i.e., for behavioural rule $b \in B$ representing the transparent mode, i.e., $F(cond(b)) \cap V_{\uparrow\downarrow} = \emptyset$, a value of $var(b)$ in the next step should be calculated from values of all variables in $cond(b)$ and $value(b)$ in the next step too. To reflect this, we adjust behavioural rule b such that every variable reference v that appear in $cond(b)$ and $value(b)$ will point to its next value v'. On the other hand, if the behavioural rule b represents the synchronous mode, the next value of $var(b)$ should be calculated from current values of $value(b)$. 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 $b_\tau \in B$ representing the transparent mode whose condition is built solely of the conditions $c_1, c_2, \ldots,$ c_n (possibly negated), for $cond(b) = c_1, \ldots, c_n, \uparrow v, c_{n+1},$ $\ldots, c_m$. Due to the order of the evaluating conditions, the b_τ rules have a priority over b (note that 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 $c_1, c_2, \ldots, c_n$ in b.

4.4 Generating Counter Automata

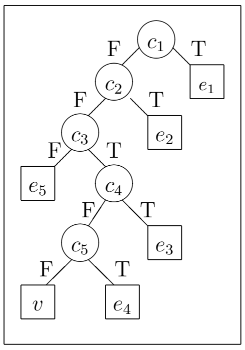
We start building the counter automaton A representing the design by defining its *set of counters* as the set of all integer-typed state variables from V—formally, wrt. the definition of counter automata, $X = \{v \mid v \in V_s,$ $T(v) = int\}$. Further, we build control locations of A based on all possible valuations of all *control state variables* in V, i.e., 1-bit state variables from the set $V_q = \{v \in V_s \mid T(v) = bool\}$. Formally, we define the set of control locations of A as $Q = \{q \mid q : V_q \to \{0, 1\}\}$.

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 counters[2]. 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.

[2]In 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.

```
if c1 then
  v <= e1;
elsif c2 then
  v <= e2;
elsif c3 then
  if c4 then
     v <= e3;
  elsif c5 then
     v <= e4;
  else
     v <= v;
else
  v <= e5;
```

(a)



(b)

$c_1 \to v := e1$
$\neg c_1, c_2 \to v := e2$
$\neg c_1, \neg c_2, c_3, c_4 \to v := e3$
$\neg c_1, \neg c_2, c_3, \neg c_4, c_5 \to v := e4$
$\neg c_1, \neg c_2, c_3, \neg c_4, \neg c_5 \to v := v$
$\neg c_1, \neg c_2, \neg c_3 \to v := e5$

(c)

Figure 2: 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.

Considering 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: (1) For each $b \in B$ with $\mathit{cond}(b) = c_1 c_2 \ldots c_n$ for some $n \in \mathbb{N}$, we (as far as possible) evaluate the enabling condition of b, i.e., we compute $\mathit{guard}^{q_1,q_2}(b) = \bigwedge_{1 \le i \le n} c_i^{q_1,q_2}$. Further, let $B_e = \{b \mid b \in B,\ \mathit{var}(b) \in V_s,\ \mathit{guard}^{q_1,q_2}(b) \neq \mathit{false}\}$ 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 $\mathit{var}(b) = v$ for each state variable $v \in V_s$. For each B_t, we perform the following steps:

1. 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 $\mathit{value}(b_v)$ where $b_v \in B_t$ and $\mathit{var}(b_v) = v$.[3] We repeat this step till all references to future values of counters disappear.

2. Based on the set of rules B_t, we create a transition $q_1 \xrightarrow{\varphi} q_2$ of A where $\varphi = (\bigwedge_{b \in B_t} \mathit{guard}^{q_1,q_2}(b)) \wedge (\bigwedge_{b \in B_t, \mathit{var}(b) \notin V_q} \alpha(\mathit{var}(b) := \mathit{value}^{q_1,q_2}(b))$ and α is a function that transforms an assignment $v := e$ to a formula $v' = e$.

4.5 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 4.2). In particular, we implemented a translation to counter automata in the input language of the ARMC tool [17] and also to integer programs in the C programming language for Blast [23]. 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 a propositional formula.

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. 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 FIFO*. We successfully verified two properties of the component: (1) the queue does not inform that *it is full* incorrectly (on four different configurations of the AsFIFO-Full), and (2) the queue does not inform that it is *empty and full at the same time* (AsFIFO-FE).

The results of our experiments are summarized in Table 3. 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 num-

[3]At 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.

Component	$\|Q\|$	$\|\delta\|$	$\|X\|$	Extraction	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 3: Experiments with extracting from VHDL and with their subsequent reachability analysis.

ber 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.

5. Conclusion and Future Research

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 Asynchronous Hardware Designs

For verifying asynchronous systems with multiple clock domains, we have introduced four original approaches. 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 approach (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*, 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.

Future Directions in Verifying CDC. Since 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. A possible solution that is worth exploring in the future is 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. 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. 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.

5.2 Verifying Parametrized Hardware Designs

To deal with parametrized hardware designs, we have presented a new, quite general and automated, approach to formally verify parametrized VHDL components. 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). We have implemented a translation scheme presented in the thesis in the prototype tool *VHD2CA*. It was successfully used together with the ARMC tool [17] for verification of several interesting properties of parametrized VHDL components, including a real-life component.

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. Another interesting research direction is to investigate possibilities of reducing the size of the automata that we generate. Since some transitions generated by the algorithm proposed before 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.

Acknowledgements. The work presented here was partially supported by the Czech Ministry of Education (research plan no. MSM 0021630528), the Czech Science Foundation (project no. GP103/10/0306), and the Brno University of Technology, Faculty of Information Technology (project no. FIT-10-S-1).

References

[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] E.M. Clarke, O. Grumberg, and D.A. Peled. Model Checking MIT Press, 1999. ISBN: 0-262-0327-8.

[4] 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.

[5] 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.

[6] D. J. Kinniment, A. Bystrov, A. Yakovlev. Synchronization Circuit Performance. In *IEEE Journal of Solid-State Circuits*, Volume 37, pp. 202–209, 2002.

[7] 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.

[8] 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.

[9] Clock Domain Crossing – Closing the Loop on Clock Domain Functional Implementation Problems. *Technical Report*, Cadence Design System, 2004.

[10] R. Ginosar. Fourteen Ways to Fool Your Synchronizer. In *Proceedings of ASYNC'03*, IEEE Computer Society, 2003.

[11] T. Kapschitz, R. Ginosar, and R. Newton. Verifying Synchronization in Multi-Clock Domain SoC. In *Proceedings of DVCon'04*, @HDL, Inc., 2004.

[12] T. Kapschitz and R. Ginosar. Formal Verification of Synchronizers. In *Proceedings of CHARME'05*, LNCS 3725, pp. 359–362, Springer Berlin / Heidelberg, 2005.

[13] 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.

[14] 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.

[15] M. L. Minsky. *Computation: Finite and Infinite Machines* Prentice-Hall International, 1967. ISBN: 0-13-165563-9

[16] H. Comon, Y. Jurski. Multiple Counters Automata, Safety Analysis and Presburger Arithmetic. In *Proceedings of CAV'98*, LNCS 1427, Springer Berlin / Heidelberg, 1998.

[17] 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.

[18] 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.

[19] S. Bardin, A. Finkel, and E. Lozes. From Pointer Systems to Counter Systems Using Shape Analysis. In *Proceedings of AVIS'06*, 2006.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

Selected Papers by the Author

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.

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.

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.

P. Matoušek, A. Smrčka, and T. Vojnar. High-level Modelling, Analysis, and Verification on FPGA-based Hardware Designs. CESNET Technical Report no. 8/2005.

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.