Verifying VHDL Designs with Multiple Clocks in SMV*

A. Smrčka[1], V. Řehák[2], T. Vojnar[1], D. Šafránek[2], P. Matoušek[1], and Z. Řehák[2]

[1] FIT BUT, Brno, Czech Republic
{matousp,smrcka,vojnar}@fit.vutbr.cz
[2] FI MU, Brno, Czech Republic
{xsafran1,xrehak,xrehak5}@fi.muni.cz

Abstract. The paper considers the problem of model checking real-life VHDL-based hardware designs via their automated transformation to a model verifiable using the SMV model checker. In particular, model checking of asynchronous designs, i.e., designs driven by multiple clocks, is discussed. Two original approaches to compiling asynchronous VHDL designs to the SMV language such that errors possibly arising from the asynchronicity are preserved are proposed. The paper also presents results of experiments with using the proposed methods for verification of several real-life asynchronous components of an FPGA-based router being developed within the Liberouter project.

1 Introduction

The most recent verification technologies for design of hard-wired ASIC-based hardware or FPGA firmware offer highly developed industrial verification tools which give hardware designers a good support to minimise errors in the design of hardware synthesised from high-level descriptions usually written in languages like VHDL or Verilog. Such tools can be basically divided into two groups—simulation and testing, and formal verification tools. Tools of the first group are focused on simulation of gate-level signal behaviour and are commonly used, according to our experience, by hardware designers as a necessary support of hardware development. Tools of the second group augment the non-exhaustive simulation approach by model checking (formal assertion-based verification) of entire Register Transfer Level (RTL) hardware description [13, 4], or checking of equivalence between an RTL description and the respective behavioural description [5, 2]. However, because of limitations caused by the state explosion problem, these tools still lack the property of being usable by verification non-experts. In the case of, otherwise highly automated, model checking tools, the reason is that intricate abstraction methods are needed to fight the state explosion. Especially, such abstraction methods must be employed carefully to avoid any critical underapproximation potentially introduced in the model being verified.

The most fundamental abstraction used in verification of hardware is the abstraction of the physical latency of a signal value change, the so-called *zero-delay*. Our recent experience gained during verification of an FPGA-based design in the Liberouter

* This research has been supported by the CESNET activity "Programmable hardware". Zdeněk Řehák has been partially supported by the Academy of Sciences of the Czech Republic grant No. 1ET408050503. Vojtěch Řehák has been partially supported by the research centre "Institute for Theoretical Computer Science (ITI)", project No. 1M0021620808. David Šafránek has been supported by the Grant Agency of Czech republic (GA CR) grant No. 201/06/1338, Aleš Smrčka and Tomáš Vojnar have been supported by the GA CR No. 102/04/0780, and Petr Matoušek by the grant GA CR No. 102/05/0723.

project [10] shows that such abstraction cannot be used for verification of some typical parts of common FPGA hardware designs. More specifically, the zero-delay abstraction is inadequate for designs controlled by clocks of two or more mutually asynchronous clock domains. Especially, functional verification of clock domain crossing (CDC) signals behaviour requires a special care. At the same time, even though hardware designers typically use some standard constructions to deal with CDC, they may be omitted by mistake or a wrong mechanism wrt. the assumptions used in the rest of the design may be used, and so there is a real need to check for possible errors related to asynchronicity. Moreover, errors introduced due to an unexpected behaviour of CDC signals cannot be easily found by standard simulation and testing tools.

The Liberouter project is aimed at the development of a high-speed network monitoring and routing hardware [8, 7] in the form of add-on cards for standard PCs. The design of these cards is based on FPGA technology. We have been employing various formal methods for verification of the design since the beginning of the project [12, 9]. In this paper, we generalise our approach [6] of a direct temporal logic-based formal verification of VHDL hardware description using Cadence SMV [13] for the case of asynchronous hardware. Moreover, although we present our approach in a framework specific for our project, we believe that it offers a general idea of how verification of asynchronous hardware can be done even in different settings.

1.1 Related Work

There are simulation and testing approaches [1] which deal with transient behaviour of designs. However, all these methods are incomplete because of the non-exhaustive nature of design behaviour analysis.

The approach of [11] requires the design with CDC signals to be transformed into a design extended with additional combinational logic which models the potential misbehaviour of CDC signals. Assertion-based verification is then applied to the resulting design. The number of combinational logic elements added in this transformation increases exponentially with the number of asynchronous clocks, thus the state explosion of the resulting design complicates the verification. Moreover, there is no simplification in the sense of automated detection of those parts of the design for which the discussed transformation is necessary for a correct verification. Detection of CDC signals is realised at the verification phase itself, hence the state explosion cannot be overcome anyway in this method. Moreover, there are no arguments showing the generality of the approach. The approach is illustrated on a simple example only and no discussion of its efficiency is given.

Our approach offers a solution based on an extension of every critical gate to incorporate the delayed behaviour. Additionally, we also introduce an approximate solution which suffers much less from the state explosion. In contrast to the work mentioned in the previous paragraph, we focus on generality of our approach. Moreover, we also evaluate the efficiency of our approach on a real case study.

1.2 Our Contribution

In this paper, the problem of the above mentioned inadequacy of the zero-delay abstraction for multi-clock designs is carefully discussed and a general verification solution for dealing with such designs is established. The proposed solution comprises a detection of the relevant parts of the design for which the zero-delay abstraction cannot be employed

and furthermore defines a way of how such design parts are transformed and verified using Cadence SMV. Besides the accurate solution, we also propose a solution based on a safe overapproximation of the reachable states. According to our practical experience, this solution is precise enough to handle various non-trivial real-life case studies while offering much better performance results. The proposed methods are demonstrated and compared on a real verification case study taken from the Liberouter project.

To the best of our knowledge, in the literature there is currently no general fully automated model checking solution for formal verification of designs controlled by asynchronous clocks. Moreover, our approach employs the Cadence SMV model checking tool which supports temporal logics (LTL, CTL). These logics are more expressive then traditional assertion languages used in many industrial verification tools.

The structure of the paper is the following. Section 2 brings a brief introduction to digital elements in digital hardware design, explains the case of situations when the synchronisation problem occurs, and presents a way of formalising elementary hardware entities in SMV. Section 3 describes precisely our methods how to find critical signal paths in the design and how to verify the system with respect to the considered properties. Section 4 illustrates our solution on a real example.

2 Formalising a Hardware Design

In this section, we shortly introduce elementary digital circuits and present our approach to their formalisation in SMV. We are mainly concerned about precise modelling which considers timing delays and unstable behaviour of the circuits. The issue is the most critical for design and verification of synchronous digital circuits. In order to prove that the design is correct using verification techniques, we have to (1) build a formal model of the circuit that reflects the examined properties including the timing behaviour, and (2) successfully verify the model using a model checker.

2.1 Preliminaries

In our work, we deal with logical circuits—discrete electronic circuits composed of basic entities like gates, flip-flops, and latches. A *gate* is a logical circuit with one or more inputs that produces an output based on the current input values. The most well-known are AND-, OR-, NOT-gates (or NAND-, NOR-gates) that are fundamental elements of every logical circuit. A gate is usually called a *combinational circuit*, or a combinational logic, as its output depends only on the current input combination [14].

Logical circuits that are able to store a value (they work like a register) are called *sequential logical circuits*. An output of a sequential circuit depends not only on its current inputs, but also on the past sequences of inputs. Formally, we can describe the behaviour of a sequential logical circuit using a finite-state machine.

We distinguish two basic kinds of sequential logical circuits—latches and flip-flops. A *latch* is a sequential circuit that continuously watches all of its inputs and changes its outputs at any time, independently of a clock signal. A *flip-flop* is a sequential circuit that normally samples its inputs and changes its outputs only at the instants determined by a clock tick. Common sequential circuits are D-latch, S-R latch, J-K flip-flop, etc.

2.2 Transient Behaviour

When dealing with the transient behaviour, we have to take into account what happens when the signal changes between two adjacent states, e.g., on a falling edge (the signal

changes from a low level to a high level), or on a rising edge (from a high level to a low level). In real circuits, changes of a signal take a nonzero time. The amount of time that the output of a logical circuit requires to change its state is called the *transition time*.

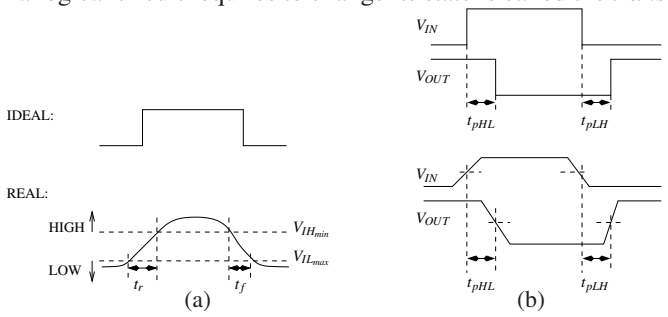


Fig. 1. (a) Transition times, (b) propagation delays

Fig. 1(a) shows the rise time t_r and the fall time t_f of a signal in a logical circuit. This time information indicates how long an output voltage takes to pass through the "undefined" region between the LOW and HIGH levels of signal.

The initial part of a transition before reaching the value $V_{IL_{max}}$, resp. $V_{IH_{min}}$, is not included in the rise- or fall-time value. Instead, it forms another parameter called a *propagation delay* t_p. The propagation delays t_{pHL} and t_{pLH} represent the amount of time that it takes for a change of the input signal to produce a change of the output signal, see Fig. 1(b). Finally, t_{pHL} is the time between an input change and the corresponding output change when the output is changing from HIGH to LOW, and t_{pLH} is the time between an input change and the output change when the output is changing from LOW to HIGH (cf. Fig. 1(a).

2.3 Synchronisation between Two Clock Domains

The transient behaviour described above is usually not considered in an abstract model obtained when a hardware design specified by VHDL is transformed into an input language of a verification tool. We call this kind of abstraction the *zero delay abstraction*. In general, omitting the transient behaviour in the abstract model can lead to incorrect results. In this section, we explain the synchronisation problem that may cause the zero delay abstraction to produce incorrect results in cases of asynchronous designs. In the next section, we propose a precise way of modelling the transient behaviour.

If there is only one clock signal in the design, we do not need to care about propagation delays and transition times. In such a case, the transient behaviour of any circuit has no influence on the other parts of the design because every signal becomes stable after the same period, and we may assure the period to be long enough for the signals to stabilise—this issue is ensured by common hardware development tools. A *synchronisation problem* occurs if two or more communicating circuits are controlled by different clocks. Fig. 2 demonstrates the synchronisation problem between two directly connected gates X and Y. Gate X is controlled by clock C1, gate Y by C2, X has two output signals A and B, B is a negated version of A. When clock C2 is enabled at time t_{err}, both signals A and B are in the process of changing. At this moment, their state is

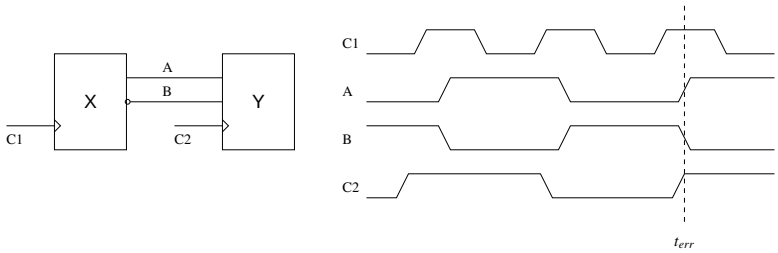


Fig. 2. The synchronisation problem between two clock domains

undefined—both signals can be read out by gate Y either as 1 or 0 because their values are not stable yet. This is the critical moment for the circuit behaviour and its modelling.

In our work, we propose a technique how to model this behaviour in order to verify properties on a real-world design. In the analysis, we use a notion of a *clock domain*. A clock domain is a part of the circuit. It is the maximal set of gates that are enabled by the same clock signal. From the point of view of synchronisation, *critical gates* are the gates on a signal path connecting different clock domains. A non-consistent behaviour can appear while reading data transferred from gate X in domain D_1 (enabled by signal C_1) at gate Y in domain D_2 (controlled by signal C_2), see Fig. 2.

To eliminate inconsistencies caused by the transient behaviour, designers typically use Gray code, mux-synchronised signals, handshake synchronizers, or asynchronous queues. Gray code is useful to guarantee a correct transfer of an integer variable whose next value differs only by one digit from the previous one. The method of multiplexer synchronised signals ensures that in one moment in time only one signal value may change. Handshake synchronizers, asynchronous queues, and other techniques not mentioned in this paper are more general. Here, our concern is how to properly model the transient behaviour in order to verify circuits where it appears.

2.4 Digital Circuits Design in VHDL and SMV

For the model checking approach, we need to specify the model formally as a finite state transition system where states represent the current signal values and transitions represent their discrete changes. For this purpose, we use the *Cadence SMV* language [13] which allows us to encode such a model. Moreover, using the SMV tool, we can prove the properties of the circuits that we are interested in. For specification of such properties, we use a temporal logic. In this section, we describe how this modelling is done. In particular, we demonstrate our technique of formalising a hardware design described by VHDL. However, the approach is not dependent on a certain language and can easily be adapted to other specification languages, e.g., VERILOG.

Each state of a transition system can be expressed as a vector of current values of signals at a particular discrete point of time. In the timing diagram depicted in Fig. 3, states are represented as columns of 0s and 1s denoting the LOW and the HIGH level of signals (HIGH corresponds to 1 and LOW to 0). For elementary circuits controlled by only one clock, we assume the *zero delay abstraction* to be used, which means abstracting from the transient behaviour. Each transition (verification step) models the *instantaneous change* of some signals with respect to their current values contained in the source state. The target state then contains the new values of the signals being changed.

A combinational logic is captured directly by the notion of states. Relations between signal values in a particular state specify a logical function of some combinational logic elements. As the transition time has no influence on the behaviour of a combinational

logic, the zero delay abstraction fits here well. The modelling of sequential logic elements is more involved. In particular, a change of an output signal value in a sequential circuit is modelled by a transition between states.

VHDL:

```
process (gate, in)
  begin
  if gate = '1' then
    out <= in;
  end if;
end process;
```

SMV:

```
do {
  if (next(gate) = 1)
    next(out) <= next(in);
  }
```

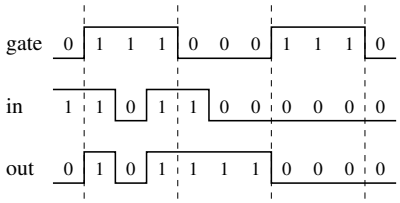


Fig. 3. Latch in Cadence SMV

In the case of a latch, any change of the `out` signal is guarded by a simple condition which requires the gate signal to have the HIGH level value. Encoding of a latch in SMV is showed in Fig. 3. An example of a trace of the respective transition system is depicted in the right-hand part of the figure. In every state in which `gate` is 1, the signal `out` has the same value as the signal `in`; otherwise, it keeps its previous value. Note that this behaviour is independent of any clock signal. Due to this asynchrony, the zero delay abstraction does not violate soundness of the model.

VHDL:

```
process (clk)
begin
  if clk'event and
    clk='1' then
      out <= in;
  end if;
end process;
```

SMV:

```
do {
  if (clk = 0 & next(clk) = 1)
    next(out) <= in;
  }
```

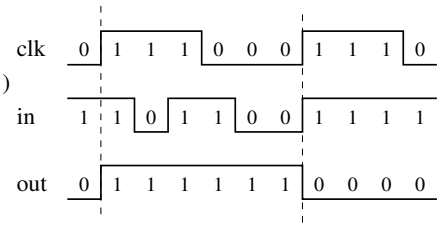


Fig. 4. Flip-Flop in Cadence SMV

In contrary to the latch case, modelling of a flip-flop is more complicated. More specifically, in the case of a flip-flop circuit, whenever the current value of the clock signal is LOW and the next value is HIGH, the next value of the output signal is set to the current value of input. In Fig. 4 there is an SMV encoding of this behaviour and an example of its trace.

Above, we have shown how we encode elementary design entities in SMV. Below, we describe our general approach of modelling compositions of these elementary entities with correct treating of the transient behaviour whenever it cannot be abstracted.

3 Modelling Asynchronous Behaviour

In this section, we propose two ways of modelling asynchronous VHDL designs in SMV which preserve errors possibly caused by the asynchronicity. In particular, we are interested in preserving *reachability of stable input and output values and state signals combinations* of the circuits. By *stable signal values*, we understand the values that are obtained after a circuit is given a sufficient time to stabilise, i.e., values that one can eventually observe when there is no change in clock signals. Moreover, the reasoning

can be generalised to preservation of sequences of stable signal values allowing one to verify complex temporal properties.

An undesirable state can happen if a sequential gate reads an unstable signal value and then it changes to a stable value. As we have already indicated, common VHDL development tools (e.g., Leonardo [3]) check that signal paths within the same clock domain are not too long wrt. the used clock frequency, and thus that the signals are given sufficient time to stabilise. If the verified system has to be connected to systems such that some input/output signal goes from one clock domain to another domain, the entire combined system should be re-checked using the methods we propose here. Thus, *we ignore the possibility of obtaining unstable signal values within one clock domain*.

We further suppose that *the only asynchronicity in the circuits we consider is due to the clock signals*. This corresponds to the assumption that the set and reset signals which may also be used to control sequential circuits are all connected together (i.e., there is just one set and/or reset signal for the whole design).

Both of the approaches we propose are based on modifying the behaviour of the so-called *critical signal paths* between two clock domains. In the first approach, we modify every gate on a critical path to make its output undefined (arbitrarily zero or one) for a single verification step whenever a change occurs. In the second approach, we introduce a special component called a *destabilizer* at the end of every critical path. This component produces an undefined output for a number of verification steps corresponding to the accumulated delay of the critical path.

We argue that the first mentioned approach enables us to detect all the possible erroneous signal combinations as described above, and at the same time, no overapproximation is (in most usual practical cases) involved. However, there is a price to be payed for this due to an increased number of state variables contributing to the state explosion problem. On the other hand, the second approach is a safe overapproximation that may work in a number of practically interesting situations in a much faster way than the first approach, but it may sometimes raise false alarms. Note that we *suffice with focusing on only the critical paths* as we suppose the design to be already checked by the above mentioned common VHDL development tools which assure us that within a particular clock domain, all the signals always stabilise before they are sensed by sequential gates.

Below, we first formalise the notion of critical paths and then explain both of the approaches we propose in detail.

3.1 Critical Signal Paths and Critical Gates

In order to precisely define the notion of critical gates, we view a particular VHDL *hardware design* in an abstract way as a triple $H = (S, C, G)$ 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 $C_\perp = C \cup \{\perp\}$.
- $G \subseteq C_\perp \times 2^S \times 2^S$ is a finite set of *gates* (combinational logic gates, flip-flops, latches). A gate is represented as a tuple consisting of its clock signal (which is $\perp$ for combinational gates), a set of input signals, and a set of output signals.

For a hardware design $H = (S, C, G)$, a *signal path* $\pi = \langle g_1 s_1 g_2 s_2 \ldots s_{n-1} g_n \rangle$ of length $n > 1$ is a connected sequence of gates and signals such that $\forall j \in \{1, \ldots, n-$

$1\} : g_j = (c_j, I_j, O_j) \in G \wedge g_{j+1} = (c_{j+1}, I_{j+1}, O_{j+1}) \in G \wedge s_j \in O_j \cap I_{j+1}$. We denote $\Pi(H)$ the set of all signal paths of H, and for a signal path $\pi \in \Pi(H)$, we denote $\Gamma(\pi)$ the multiset of all the gates which appear in π and $\Sigma(\pi)$ the set of all signals in π. For a path $\pi = \langle g_1 s_1 g_2 s_2 \ldots s_{n-1} g_n \rangle$, we call $\gamma_i(\pi) = g_1$ the input gate, $\gamma_o(\pi) = g_n$ the output gate, $\sigma_i(\pi) = s_1$ the input signal, and $\sigma_o(\pi) = s_{n-1}$ the output signal.

We partition the set of gates G of a hardware design $H = (S, C, G)$ into subsets called *clock domains* that contain gates driven by the same clock signal. For $c \in C_\perp$, the clock domain is $D_c = G \cap (\{c\} \times 2^S \times 2^S)$.

The set R_c of *gates critical wrt. a domain* D_c, $c \in C$, is the set of gates which occur on signal paths leading to D_c and that are connected to the gates in D_c via combinational gates only. Equivalently, for a domain D_c, critical gates are all the gates on the signal paths that start by a sequential gate lying in a different clock domain (including this sequential gate) and lead via combinational gates to a sequential gate in D_c (excluding this terminal gate). Formally, $R_c = \{g_1 \in G \setminus D_c \mid \exists n > 1\ \exists s_1, \ldots, s_{n-1} \in S\ \exists g_2, \ldots, g_{n-1} \in D_\perp\ \exists g_n \in D_c : \langle g_1 s_1 g_2 s_2 \ldots s_{n-1} g_n \rangle \in \Pi(H)\}$ for a hardware design $H = (S, C, G)$. The set $R(H)$ of *critical gates of* H is then simply the union of all the gates critical wrt. the particular domains of H, i.e., $R(H) = \bigcup_{c \in C} R_c$.

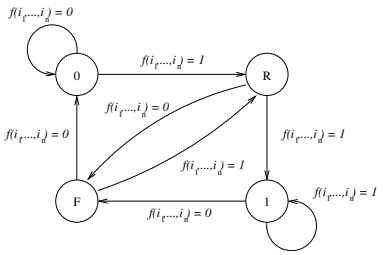
Finally, a *critical signal path* of length $n > 1$ in a hardware design $H = (S, C, G)$ is a signal path $\rho = \langle g_1 s_1 g_2 s_2 \ldots s_{n-1} g_n \rangle \in \Pi(H)$ that consists of critical gates, i.e., $\forall i \in \{1, ..., n-1\} : g_i \in R(H)$, and goes from one clock domain to another one, i.e., $g_1 \in D_{c_1}, g_2 \ldots g_{n-1} \in D_\perp, g_n \in D_{c_2}, c_1 \neq c_2, c_1 \neq \perp, c_2 \neq \perp$. We denote $\rho(H)$ the set of all critical signal paths in H.

3.2 Extending All Critical Gates

We now discuss in detail the approach when we extend the behaviour of every gate on a critical path to make its output random whenever there is a change of the stable value in its output. The fact that we make the output 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 sequential gate on such an edge, one cannot predict its value. Note that when there is no change in the output, no modification is necessary.

The basic principle of our transformation is the following. To model the impact of rising and falling edges in the output of critical gates, we replace every output of every critical gate by a new state signal—we call it a *delayed output*. The values of a delayed output are given by the states of the finite automaton in Fig. 5(a). The arcs represent the original (zero-delayed) output signal defined by a function $f(i_1, \ldots, i_n)$ where i_1, ..., i_n are input signals of the gate. For example, when the delayed output is 0 and the original output changes to 1, the automaton goes to R, and the delayed output becomes R (i.e., "rising"). Only then, provided the original output does not change, it transfers to 1. Similarly, for a change of the original output from 1 to 0, the delayed output goes from 1 to F ("falling") and then changes to 0.

As an example, let us consider an inverter (a NOT-gate with the function $f(A) = \neg A$ where A is the only input signal) which is a critical gate. We extend this gate with the delayed output as shown in the table in Fig. 5(c). In the table, Y' is the future value of the delayed output. Note that a gate that was originally a combinational one becomes a gate with a state now.



(a)

A	$Y = f(A)$
0	1
1	0

(b)

A	$f(A)$	Y	Y'
0	1	0	R
0	1	R	1
0	1	1	1
0	1	F	R
1	0	0	0
1	0	R	F
1	0	1	F
1	0	F	0

(c)

Fig. 5. (a) The finite automaton implementing a delayed output with the rising and falling edges of signals, (b) the transition table of the zero-delayed NOT-gate, (c) delayed extended NOT-gate

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).

To continue with our example, suppose that the input of the inverter (NOT-gate) is a signal controlled by a clock $C1$ and that both the input and the output are sensed by some sequential gate controlled by a clock $C2$. When the gates are modelled as zero-delayed, cf. Fig. 6(a), we will never see that the sequential gate can sense both the signals as equal (which we may suppose to cause real erroneous situations). This will become visible in our model as depicted in Fig. 6(b)—see x-values depicted as crosses in the figure.

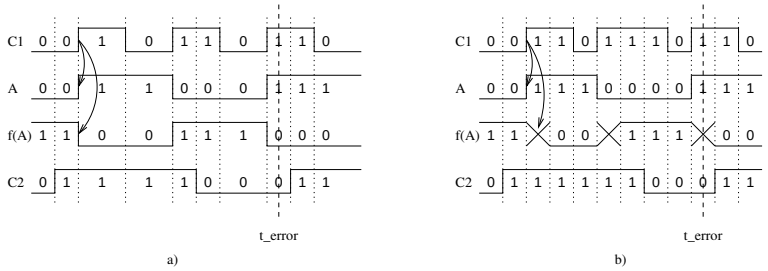


Fig. 6. The input and output of the NOT-gate sensed as (a) zero-delayed and (b) delayed

A problem with the above extension of gates arises when there exist two paths linking some sequential gates via combinational gates only such that one of them stays within a single clock domain, the other one goes from one domain to a different one, and the two paths intersect each other. More precisely, the problem arises when there exist signal paths $\pi_1 = \langle g_{1,1}s_{1,1}\dots s_{1,n_1-1}g_{1,n_1}\rangle$, $\pi_2 = \langle g_{2,1}s_{2,1}\dots s_{2,n_2-1}g_{2,n_2}\rangle \in \Pi(H)$ of a hardware design $H = (S,C,G)$ for $n_1,n_2 > 1$ and some domains $c_1,c_2,c_3 \in C$, $c_2 \neq c_3$ such that $g_{1,1},g_{1,n_1} \in D_{c_1}$, $g_{2,1} \in D_{c_2}$, $g_{2,n_2} \in D_{c_3}$, $\forall i \in \{1,2\}\ \forall j \in \{2,\dots,n_i-1\} : g_{i,j} \in D_\perp$, and $\Gamma(\pi_1) \cap \Gamma(\pi_2) \neq \emptyset$. In such a situation, we need to extend the gates in $\Gamma(\pi_1) \cap \Gamma(\pi_2)$ for the path π_2, but to keep their original function within π_1. To achieve this, before the above described extension, we pre-process the circuit by replacing every gate $g_{2,i} = (\{\perp\},I,O) \in \Gamma(\pi_1) \cap \Gamma(\pi_2)$ within the path π_2 by two gates $g'_{2,i} = (\{\perp\},I,O \setminus \sigma(\pi_2))$ and $g''_{2,i} = (\{\perp\},I,O \setminus \sigma(\pi_1))$ with the same behaviour. We call the new gate g'' *a duplicate*. For further analysis, let n_{dup} denote the number of new output signals produced by duplicates.

Extending Simple Combinational Logic Gates in SMV. So far, we have described the main principle of our technique of modelling asynchronous VHDL designs such that errors possibly arising due to the asynchronicity are preserved. We now have a look at how to apply this principle on transforming concrete gates from VHDL to SMV. We start with the simple case of combinational logic gates.

Recall the zero-delayed model of a given design described in Section 2.4. In such a model, circuits of combinational logic gates (NOT, AND, OR, NAND, NOR) are translated to SMV simply in the form of logic expressions. Now, we have to (1) define state-bearing *delaying modules* for every type of a critical combinational gate that appears in our design, (2) instantiate these modules (one instance for each particular critical combinational gate), and (3) change the interconnection of the original zero-delayed model such that the delayed outputs are used instead of the zero-delayed ones for every critical combinational gate.

A *state-bearing delaying module for a given combinational gate* has the same input and output signals as the original gate, but the output is computed according to the way described in the previous section. To implement the delaying functionality with the interleaved random signals corresponding to the automaton from Fig. 5(a), the module has an internal state variable in which we remember the output of the original gate computed according to its function (NOT, OR, ...) and we send it to the new output with a delay of one step (if there is no change in it). To detect the changes in signals, we may conveniently use the possibility of referring to the next values of signals offered by SMV. To illustrate this construction on a concrete example, we give below a delaying module for a NAND gate (described in the SMV syntax).

```
module delayed_nand(out, in1, in2) {
  input in1 : boolean;
  input in2 : boolean;
  output out : boolean;
  orig_out : boolean;
  -- the original NAND function
  next(orig_out) := ~(next(in1) & next(in2));
  -- the delayed output
  if (orig_out = next(orig_out)) next(out) := orig_out;
  else next(out) := {0,1}; -- a random choice
}
```

Then, if there is used, e.g., an assignment `z := w | ~(x & y);` somewhere, we declare a new signal for the delayed output of the NAND over `x` and `y`, a new instance of the delayed NAND computing this delayed signal, and we use this signal instead of `~(x & y);`. The construction is shown below with the delayed output of the NAND sent to a delayed OR module whose implementation is very similar to the delayed NAND and is not given here due to space limitations.

```
nand_output : boolean;
nand_module : delayed_nand(nand_output, x, y);
or_module : delayed_or(z, w, nand_output);
```

Extending More Complex Gates in SMV. More complex gates including flip-flops, latches, and complex combinational gates like multiplexers are modelled as separate SMV modules. We now have to create duplicates of such modules, extend them by new internal signals to hold the original output, and define the outputs of these new modules as delayed versions of the original outputs (interleaved with the random phases) much like in the above case of simple combinational gates. For instance, a delayed D flip-flop could then look as shown below.

```
module delayed_D(set, reset, in, clk, delayed_out) {
  input set : boolean; input reset : boolean;
  input in : boolean;  input clk : boolean;
  output delayed_out : boolean; -- the delayed output
  orig_out : boolean;     -- the original output
  do {                          -- an initialisation phase
      if (set) init(orig_out) := 1;
      else if (reset) init(orig_out) := 0;
      else if (clk) init(orig_out) := in;
      init(delayed_out) := init(orig_out);
  }
  do {                          -- computing the original output
      if (next(set)) next(orig_out) := 1;
      else if (next(reset)) next(orig_out) := 0;
      else if (~clk & next(clk)) next(orig_out) := in;
  }
  ------- the delay-based extension -- computing the delayed output -------
  if (orig_out = next(orig_out)) next(delayed_out) := orig_out;
  else next(delayed_out) := {0,1}; -- a random choice
}
```

A Justification of the Construction. The *extension of only the critical gates* is justified by our assumption that common VHDL development tools are used to check that in all single clock domains, all signals have always enough time to stabilise before being sensed by sequential gates. Moreover, we suppose that input and output signals of the entire checked design will be used within the same time domain as the gates which consume these signals.

As for the extension of critical gates, the modification makes their output *non-deterministic for a single verification step* if the changed input would lead to a change in the original output. 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 in the output, no extension is needed. An interesting situation is when there is a change in 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). 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 peak or drop 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 underapproximation or overapproximation 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 first consider the case of *two critical paths with a different length* (a generalisation 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 stabilise 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 keep the random phase longer than one step per gate in ρ_1 in order to obtain the desirable combination of two x-values at the ends of both paths. A similar reasoning can then be employed in the case of *two equally long*

paths. If the accumulated delay of one of the paths was longer than the other one, we would need to exclude the possibility of obtaining two undefined results. However, we suppose that such conditions do not arise (which is usually the case and which can be checked statically given the design and the descriptions of the used gates).

3.3 Extending Signal 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 gate. This method is rather precise but may cause a significant state-space explosion due to the number of the newly introduced state signals. Below, we try to avoid this explosion by introducing a less precise, approximate model that can, according to our experience, still be sufficient in many practical cases. In this approach, we do not extend every single critical gate, 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 overapproximate in the new approach, let us summarise how the process of stabilisation of a signal $\sigma_o(\rho)$ in a critical signal path ρ looks like in the previous approach when we extend every critical gate by the delaying and randomising phase. In that case, a critical gate can be viewed as a generator of stable and unstable values. If more critical gates are sequentially connected (they all appear in the same critical signal path ρ), the unstable values are propagated through all critical gates on the path, and every gate delays its new output value. The new defined value of the signal $\sigma_i(\rho)$ influences the signal $\sigma_o(\rho)$ after the delay equal to the sum of delays of all gates on ρ without the last gate $\gamma_o(\rho)$. When $\sigma_i(\rho)$ changes its value, it can cause a temporary instability—the adjacent gates switch their output value through a rising or falling edge when the value of the signal is not unambiguously defined, 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 $L = |\Gamma(\rho)| - 1$, i.e., unstable values of the signal $\sigma_o(\rho)$ can occur in at most L steps.

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 output signal of a critical signal path ($\sigma_o(\rho)$) where x-values can become visible. We create one destabilizer for every set of critical signal paths having the same output signal. The destabilizer starts to generate x-values if one of the input signals of these signal paths changes its value.

Formally, a *destabilizer over a critical path* ρ in a design $H = (S, C, G)$ is a gate $\delta_\rho = (\bot,\ \alpha(\rho) \cup \{\sigma_o(\rho)\},\ \{\omega\})$ to be added into G where $\alpha(\rho) = \{\sigma_i(\rho') \mid \rho' \in \rho(H) \wedge \sigma_o(\rho') = \sigma_o(\rho) \wedge \gamma_o(\rho') = \gamma_o(\rho)\}$ is the set of input signals to be monitored (it is the set of input signals of all critical signal paths sharing the output with ρ) and $\omega \notin S \cup C \cup \{\bot\}$ is a new unique signal representing the output of δ_ρ. The original output signal $\sigma_o(\rho)$ of the given critical path ρ (and of the adjoining paths) becomes an input of δ_ρ and is sent to the output of δ_ρ after the phase of instability implemented by δ_ρ is over. Apart from introducing δ_ρ, we have to change the gate originally connected to ρ, i.e., $\gamma_o(\rho)$, such that it senses the output of δ_ρ. In particular, if $\gamma_o(\rho) = (c, I, O)$, we replace it by $\gamma'_o(\rho) = (c,\ (I \setminus \{\sigma_o(\rho)\}) \cup \{\omega\},\ O)$.

The behaviour of a destabilizer δ_ρ is defined by the finite automaton shown in Fig. 7—for brevity, the automaton is described with one bounded counter whose pos-

sible values are not included directly in the state-transition control. Let $\nu(\alpha) \neq \nu'(\alpha)$ for a set of monitored signals α denote that some signal in α 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* verification steps where *L* is the number of critical gates in the longest critical signal path which the destabilizer is connected to, i.e., *L* is the maximum of $|\Gamma(\rho')| - 1$ for $\rho' \in \rho(H) : \sigma_o(\rho') = \sigma_o(\rho) \wedge \gamma_o(\rho') = \gamma_o(\rho)$.

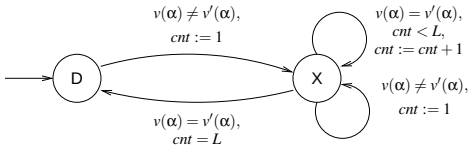


Fig. 7. The automaton describing the behaviour of a destabilizer

Destabilizers in SMV. 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,...,in_n` to be connected to the monitored inputs of the covered critical paths, the input signal `out` (the original output to be delayed), and the new output signal `omega`.

```
module Destabilizer(in_1, ..., in_n, out, omega) {
  output omega : boolean;
  input out : boolean;
  input in_1 : boolean;
  ...
  input in_n : boolean;
  cnt : 0..L;         -- L is a constant value   L = |Γ(ρ)| − 1
  init(cnt) := 0;
  init(omega) := init(out);
  next(cnt) := case {
    -- one of the monitored signals is changing
      (in_1!=next(in_1)) |
      ...                |
      (in_n!=next(in_n))          : 1;     -- to state X
    -- the counter reaches the maximum and all monitored signals hold
      cnt=L & (in_1=next(in_1)) &
               ...              &
             (in_n=next(in_n))    : 0;     -- to state D (cnt=L-1)
    -- the counter is in (0;L) and all monitored signals hold
      cnt>0 & (in1=next(in_1)) &
               ...             &
               (in2=next(in_n))    : cnt+1; -- to state X
    -- all signals are stable
      1                            : 0;     -- to state D
  }
  next(omega) := case {
      next(cnt)=0 : next(out);  -- cnt==0: propagate a defined value
      next(cnt)>0 : {0,1};      -- cnt>0: output x-value
  }
}
```

To illustrate how a destabilizer is connected to the rest of a modelled design, let us consider a signal `o` whose stable value is computed as a function `comb` implemented by a combinational logic with inputs $\alpha = \{$ `s_1`, ..., `s_n` $\}$ and which is at the end of a critical path. Let `Z` be the output gate consuming `o`. `Z` is a sequential gate that is in

a different clock domain than the gates from which the inputs s_1, ..., s_n are taken. In SMV, this would correspond to the code fragment below.

```
o := function_comb(s_1, ..., s_n);
Z_m : Z(..., o, ...);
```

To introduce a destabilizer, we define a new delayed output omega, instantiate a destabilizer with delayed_o as its output, connect the original output o as an input of the destabilizer, and replace the original output at the input of Z with omega.

```
o := function_comb(s_1, ..., s_n);
omega : boolean;
destabil_m : Destabilizer(s_1, ..., s_n, o, omega);
Z_m : Z(..., omega, ...);
```

A Justification of the Construction. We are interested in proving 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 overapproximates 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 gates in critical signal paths, it takes at most $L = |\Gamma(\rho)| - 1$ steps to stabilise the output signal if the input signal of any critical path changes (provided ρ is the longest path). A destabilizer produces x-values for L steps if any of the input signals changes. Thus, the destabilizer method will generate all the combinations of signals to be sensed and become stable as in the method based on extending all gates in critical signal paths and may be even more. Therefore, it is a safe overapproximation of the extension of all gates in critical signal paths.

However, if a model checker returns a counterexample in a model using destabilizers, we cannot be sure if it reflects a possible behaviour of the real system. In such a case, we must use a more precise 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 our future work.

We said that destabilizers often save a number of state variables compared to the method of extending all critical gates. Let us examine when this approach is efficient wrt. the number of state variables. The method based on extending all gates in a critical signal path creates one new binary state variable per a critical gate (for a critical signal path $\rho \in \Pi(H)$, this means $|\Gamma(\rho)| - 1$ new variables) plus n_{dup} state variables for duplicated state signals—we mean the duplication due to the case where $\rho \in \rho(H), \pi \in \Pi(H), \pi \notin \rho(H), \Gamma(\rho) \cap \Gamma(\pi) \neq \emptyset$. One destabilizer can replace the extension of all critical gates that is needed in the first method on more than one critical signal paths. The number of gates in these critical paths is $\lambda = |G_d| - 1$ for $G_d = \bigcup_{i=1}^{n} \Gamma(\rho_i)$ where $\sigma_o(\rho_1) = \sigma_o(\rho_2) = \cdots = \sigma_o(\rho_n)$, $n \geq 1$. The method of signal path extension adds to the system three types of variables: (a) a binary variable of the new output—the ω signal, (b) a counter of unstable values of the range $[0;\lambda]$ (i.e., the size of the counter is $\lceil log(\lambda + 1) \rceil$), and (c) n_{div} duplicated state signals for the destabilizer due to the division of input gates (which may appear when combining the method of destabilizers with extending all critical gates as explained in the next paragraph). The method using destabilizers pays off if $\lambda + n_{dup} > \lceil log(\lambda + 1) \rceil + 1 + n_{div}$, hence in the case of $n_{dup} = n_{div} = 0$, we get $\lambda \geq 5$.

Combining Both Methods. To achieve a satisfying ratio between a model accuracy and its state space size, we are able to combine both proposed methods in one model. We are interested in behaviours when an unstable value is registered and propagated further to the design. Both methods are able to stabilise the signal with the same delay (the delay of L steps). Therefore, we can apply both methods on different critical signal paths. By selecting which of the methods should be used on a certain critical signal path, we can fine-tune the verification process by trading the accuracy of the result (due to the overapproximation by destabilizers) for the model complexity (the extension of all critical gates). Such a combination is safe if we avoid the specific case when both methods influence each other as discussed below.

Consider a circuit where two critical signal paths begin in the same gate and continue with a different signal, i.e., $\rho_1, \rho_2 \in \Pi(H)$, $\rho_1 \neq \rho_2$, $g = \gamma_i(\rho_1) = \gamma_i(\rho_2)$, and $\sigma_i(\rho_1) \neq \sigma_i(\rho_2)$. When we use the extension of all critical gates on the first signal path and the destabilizer on the second path, the first method extends the gate $g = \gamma_i(\rho_1)$ with a delayed output. However, we must also preserve its zero-delayed behaviour for the second method, and so the shared starting gate g must be *divided* into two separate gates g_1 and g_2 such that if $g = (c, I, O)$, then $g_1 = (c, I, O \setminus \{\sigma_i(\rho_2)\})$ and $g_2 = (c, I, O \setminus \{\sigma_i(\rho_1)\})$. Let n_{div} be the number of duplicated state signals caused by the division of all such gates in the model.

A similar problem appears if two critical paths with an application of both methods share some of the intermediate gates. For every pair of the critical signal paths $\rho_1, \rho_2 \in \Pi(H)$, $\rho_1 \neq \rho_2$, and all shared gates $\Gamma_{1,2} = (\Gamma(\rho_1) \setminus \{\gamma_i(\rho_1), \gamma_o(\rho_1)\}) \cap (\Gamma(\rho_2) \setminus \{\gamma_i(\rho_2), \gamma_o(\rho_2)\}) \neq \emptyset$, we have to *duplicate* every shared gate $\forall g \in \Gamma_{1,2}$ and every common signal $\forall s \in \Sigma(\rho_1) \cap \Sigma(\rho_2)$.

4 Experiments

We tested our approach on two asynchronous queues from the libraries of the Liberouter project—namely, `asfifo-bram` and `asfifo-dist`. Each of the components uses a different type of memory and has a slightly different control part. For `asfifo-bram`, we checked the property that the control part does not allow to rewrite unread data with new data. For `asfifo-dist`, we checked that the component correctly sets the so-called status data on its output which informs about the saturation of the queue.

Table 1 shows results of our experiments. Suffixes in the first column mean the following: (i) *no-check* is the case when no extension is performed, (ii) *all-gates* is the case when all critical gates are extended, (iii) *destabil* means that destabilizers are used instead of extending all the gates. Column *vars no.* gives the number of binary state variables, *time* is the verification time, and *mem* is the number of allocated BDD nodes.

Table 1. Verification results

case	vars no.	time	mem
asfifo-bram-no-check	36	4.83 s	132607
asfifo-bram-all-gates	220	inf.	inf.
asfifo-bram-destabil	44	37.52 s	2446796
asfifo-dist-no-check	44	4550.07 s	9861121
asfifo-dist-all-gates	48	30556.7 s	25878822

There are eight critical signal paths in `asfifo-bram` and two critical paths in `asfifo-dist`. We can see from the number of state variables in `asfifo-bram` that there are many critical gates to extend when extending all critical gates (the number of state variables increases to 220) whereas the destabilizer-based approach found only two places

for a destabilizer (which rapidly decreases the extensions needed). For `asfifo-dist`, we can see a big difference in time and the number of BDD nodes used for a slight difference in the number of state variables. Such a contrast is caused by the nondeterminism in the model—one random value {`0,1`} divides the further state-space exploration into two directions (the first for value 0, the second for 1).

5 Conclusion

We have introduced two original approaches to modelling asynchronous hardware designs using the input language of the commonly employed Cadence SMV model checker. One of these approaches is quite precise, but may contribute to the state explosion problem in a significant way. The other approach can be much more efficient as it is based on an overapproximation of the reachable states. The approach is, however, still precise enough to allow one to prove interesting properties on various real-life hardware designs as we have illustrated by our experiments. Both of these methods may be modified to be used together with a different model checker than Cadence SMV and represent a contribution to the state-of-the-art in verifying hardware designs by allowing one to deal with asynchronous circuits.

References

1. P. Rashinka et al. *System-on-a-chip Verification*. Methodology & Techniques. Kluwer, 2001.
2. R.K. Brayton et al. VIS: A System for Verification and Synthesis. In *Proc. of CAV'96*, LNCS 1102, 1996. Springer.
3. Mentor Graphics. *Leonardo Synthesis*, 2005.
4. Mentor Graphics. *0-In Formal Verification Data Sheet*, 2006.
5. Mentor Graphics. *Formal Pro Data Sheet*, 2006.
6. J. Holeček, T. Kratochvíla, V. Řehák, D. Šafránek, and P. Simeček. Verification Process of Hardware Design in Liberouter Project. Technical Report 5/2004, CESNET, 2004.
7. J. Kořenek, T. Pečenka, and M. Žádník. NetFlow Probe Intended for High-Speed Networks. In *Proc. of FPL'05*. IEEE Computer Society, 2005.
8. J. Kořenek, P. Zemčík, and T. Martínek. FPGA-Based Platform for Network Applications. In *Proc. of DDECS'05*. University of West Hungary, 2005.
9. T. Kratochvíla, V. Řehák, and D. Šafránek. Formal Verification of a FIFO Component in Design of Network Monitoring Hardware. In *Proc. of CESNET 2006 Conference*, 2006.
10. Liberouter Project Homepage. http://www.liberouter.org.
11. T. Ly, N. Hand, and Ch. Ka kei Kwok. Formally Verifying Clock Domain Crossing Jitter Using Assertion-Based Verification. In *Proc of DVCon'04*, 2004.
12. P. Matoušek, A. Smrčka, and T. Vojnar. Modeling, Analysis, and Verification of SCAMPI2. Technical Report 8/2005, CESNET, 2005.
13. K.L. McMillan. *Cadence SMV Manual*, 2006.
14. J. F. Wakerly. *Digital Design: Principles and Practices*. Prentice-Hall, 3rd edition, 2001.