Verifying Parametrised Hardware Designs Via Counter Automata*

A. Smrčka and T. Vojnar

FIT, Brno University of Technology,
Božetěchova 2, CZ-61266, Brno, Czech Republic
{smrcka,vojnar}@fit.vutbr.cz

Abstract. The paper presents a new approach to formal verification of generic (i.e. parametrised) hardware designs specified in VHDL. The proposed approach is based on a translation of such designs to counter automata and on exploiting the recent advances achieved in the area of their automated formal verification. We have implemented the proposed translation. Using one of the state-of-the-art tools for verification of counter automata, we were then able to verify several non-trivial properties of parametrised VHDL components, including a real-life one.

1 Introduction

Modern hardware description languages (HDL) such as VHDL or Verilog allow digital hardware to be designed in a way quite close to software programming. These languages offer many features whose use constitutes a challenge for the current formal verification technologies. One of such challenges is the possibility of *parametrisation* of the designed hardware components by values from a domain that is not bounded in advance. Parametrisation is widely used, e.g., when creating libraries of re-usable hardware components.

In this paper, we propose a novel way of verifying parametrised 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 [11] 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 bit-vector (i.e. 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

* This work was supported by the project CEZ MSM 0021630528 *Security-Oriented Research in Information Technology* of the Czech Ministry of Education, by the project 102/07/0322 of the Czech Grant Agency, and by the CESNET activity "Programmable hardware".

K. Yorav (Ed.): HVC 2007, LNCS 4899, pp. 51–68, 2008.
© Springer-Verlag Berlin Heidelberg 2008

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 concentrate on verifying that certain bad configurations specified by a boolean VHDL expression (which we call an error condition) over bit as well bit-vector variables is not reachable. We have built a simple prototype tool implementing 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 [10], we have already been able to verify several non-trivial properties of parametrised VHDL components, including a real-life component implementing an asynchronous queue designed within the Liberouter project (which aims at designing new network routing and monitoring systems based on the FPGA technology) [13,9].

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., [6,18,1,15,8,4]). 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 [3,7,2]. In this work, we get inspired by the spirit of these works and try to apply it in the area of verifying generic (parametrised) 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.

Plan of the paper. In Section 2, we introduce some basics of VHDL, we comment on the VHDL constructions that we do not support, and explain the way we pre-process VHDL for the further transformations. We also introduce the notion of counter automata. In Section 3, we provide a translation from (simplified) VHDL to a certain form of intermediate behavioural rules. In Section 4, we present a translation from the intermediate format to counter automata. Section 5 comments on the reachability properties that we verify and on the way we facilitate their checking. In Section 6, we discuss our experimental results. Finally, in Section 7, we conclude and briefly discuss possible future improvements of our approach.

2 Hardware Design and Counter Automata Basics

2.1 Hardware Design

Nowadays, most of the digital hardware development is not done on the level of particular gates or transistors, but rather on the more abstract *register transfer level* (RTL). There are several languages for RTL hardware design, also known as *hardware description languages* (HDL), out of which the most widely used are VHDL and Verilog. A design specified in such a language is an input for

hardware synthesis tools, and also for hardware simulation or verification tools. A process called *synthesis* transforms a generic RTL description of a system to the gate/transistor level of a concrete electronic circuit description. Such a description serves as an input for the further production of an integrated circuit (through the so-called place&route process) or as a configuration program for field programmable gate arrays (FPGA) if they are used to implement the system.

We build our counter automata-based models from the RTL level description via an *intermediate behavioural model*. This model cannot be created from the gate level as on that level the parametrisation of the system is lost—all the parameters are already instantiated. Moreover, in our model, we are only interested in the logical behaviour of the system, not in details such as propagation delays of the gates or the set of concrete hardware elements used to physically implement the given system.

Although VHDL and Verilog are different languages, their main expressive means are quite similar from our point of view of building a counter automaton model from an RTL hardware description (and running a verification on the counter automaton). That is why, in this paper, we will discuss only the VHDL language, which, moreover, has a better support for parametrised designs.

Hardware Design in VHDL. In VHDL [11], a more complex hardware system is described in a modular way using *components*. A component is described by a definition of its *interface* and its *body*. The interface defines the inputs and outputs of a component as well as its parameters which can make the component generic. The body of a component, also known as an *architecture*, consists of a declaration of internal variables and a collection of the so-called *parallel statements* describing the behaviour of the component.

VHDL offers two types of specifying the design of an architecture—structural and behavioural. Within a structural description, we view a digital circuit as a composition of objects that may be composed of other smaller objects. In terms of the parallel statements, this approach is based on using statements of *instantiations of subcomponents* and *parallel assignment statements* (e.g., `even <= not(a1 xor a2 xor a3);`[1]). On the other hand, the behavioural approach directly describes the desired functionality of a component using the parallel statement `process` that is specified as a *sequence of statements* like *sequential assignments* or *conditionals*. We have to, however, note that sequential statements in VHDL 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 under different circumstances (we will get to this issue closer later on).

Since there is no way how to efficiently synthesise a hardware design from complex behavioural requirements, the behavioural description is widely used for a low-level description of parts of a system (e.g., logic functions, simple

[1] From a logical point of view, a variable such as `even` represents a symbolic name for the expression assigned to it only.

registers, counters), while the structural description is used for building more complex components or the entire system.

Transparent and Synchronous Mode. The so-called *transparent* and *synchronous* modes of hardware gates substantially influence the output of the gate. For example, let us have two gates connected in a cascade. If both gates work in the transparent mode (such gates are known as *latches*) and the input changes its value, the first gate instantly propagates the input to its output (the input of the second gate), and the same value propagation happens at the second gate. The result of the transparent mode is that the change of the input of the first gate instantly changes the value of the output of the second gate. Conversely, if both gates work in the synchronous mode (such gates are known as *flip-flops*), they propagate their inputs to the output one step at a time—the change of the input values of the first gate changes its output after one clock period, but this still does not immediately influence the output of the second gate (its output is changed only after another clock period). Let us add that some gates may be operated both as latches as well as flip-flops depending on some of their control inputs.

Not Considered VHDL Constructions and Behaviour. VHDL is a very rich specification language, and we do not cover it fully. However, most of the restrictions that we describe below correspond to constructions which are in theory possible, but are usually not used, represent undesirable design practices, are often not even synthesisable, or modern synthesis tools [12,14] at least issue warnings when they are used.

First, we do not support VHDL functions, procedures, delay information, and asserts which serve for a test-bench specification of the designed hardware and do not have an influence on the behaviour of the hardware.

Next, we disallow cyclic assignments in the transparent mode in a sequential description of a behaviour (e.g., `q <= not(reset and not(set and q))`, or `if (reset = '1') then a <= b; b <= a; elsif`).[2] Such assignments would complicate our constructions significantly, and in practice, they are anyway undesirable as they lead to a possible oscillation of the signals.

We concentrate on analysing reachable stable states of hardware components only. A *stable state* is a state which does not change until one or more input variables change their values. Unstable states arise due to transition and propagation delays of real gates changing their stable states (cf. Fig. 1). In general, even when we are interested only in stable states, if we do not consider unstable states at all, there is a risk that we will not capture flaws caused by reading and registering unstable values. Such a flaw can be caused either (i) by a signal path that is too long wrt. the clock frequency used, or (ii) by an asynchronous exchange of signals between two clock domains. However, the need to deal with the former issue is eliminated simply by taking into account the capabilities of standard synthesis tools. These tools automatically check that the delay arising

[2] A sequential gate works in the transparent mode when its output is controlled only by the level of the input signals.

in the longest signal path of a given circuit is safe wrt. the clock frequency used. The latter issue is a little more complicated but it is still usually solvable by using simple static analysis to check whether the given circuit uses proper synchronisation approaches (like Gray coding) for all clock domain crossing signals [17]. Hence, below, we do not consider unstable states any further.

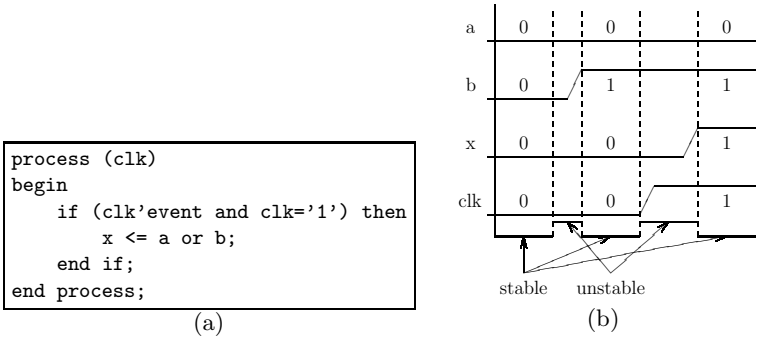

Fig. 1. (a) The source code of a simple component and (b) an example of a timed diagram of its behaviour illustrating the notion of stable and unstable states

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

2.2 Simplifying VHDL Code

To avoid a very complex direct transformation from the rich VHDL language to the intermediate behavioural model introduced in Section 3 (which is then translated to counter automata in Section 4), we first simplify a VHDL source code to a form which is much simpler for all the subsequent steps.

As we mentioned before, VHDL components contain input/output ports, parameters, and internal variables—here, we consider all of them simply as *variables*. VHDL provides two basic types of variables: *1-bit (boolean) variables* and *arrays (vectors) of bits*. Further, there is also a possibility of user-defined structured types, but they are used as a form of syntactic sugar only. Therefore, before any further steps, we decompose structured variables to their elements. Similarly, if a bit vector variable is accessed bit-wise (i.e. there is at least one statement in the considered code that accesses single bits of the vector at a time), we replace the vector variable with its boolean components (if we had not disallowed the bit-wise access to parametrised-size vectors, we would have had to use a complex arithmetics to mask out the particular bits—e.g., to get a bit

value at position p in the bit vector represented by an integer value n, we could use the expression (n `div` 2^p) `mod` 2). The remaining vectors may then easily be mapped to counters of counter automata (whereas all 1-bit variables will be a part of their control states).

Further, we also remove all structural descriptions of circuits and replace them by the corresponding behavioural description (in a way similar to macro expansion in the C programming language). This can easily be achieved by unfolding (or flattening) of the structural description taking into account that a structural description simply describes from which subcomponents a given component is build of, what are the values of parameters of the subcomponents, and how the input/output ports of these subcomponents are connected to the input/output ports of the component and/or to each other (which is done via the internal variables of the component). We substitute references to the subcomponents by their behavioural description, connect their input/output ports to the internal variables of the component (and/or its input/output ports), and substitute parameters of the subcomponents by the appropriate arguments (which may be parameters of the component being processed).

Next, we transform the code such that the only statements that will remain (and that we will have to consider in the further steps) are the following:

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

To this end, we rewrite any other statements to one or more assignment and/or conditional statements of the above form. In particular, this is the case of the VHDL *selected assignments* and `case` statements (cf. Fig. 2). Moreover, it is also the case of the VHDL `for` loops as we assume that they cannot be performed over parametric bit vectors—otherwise, we would have to model their effect by special purpose loops in our counter automata.

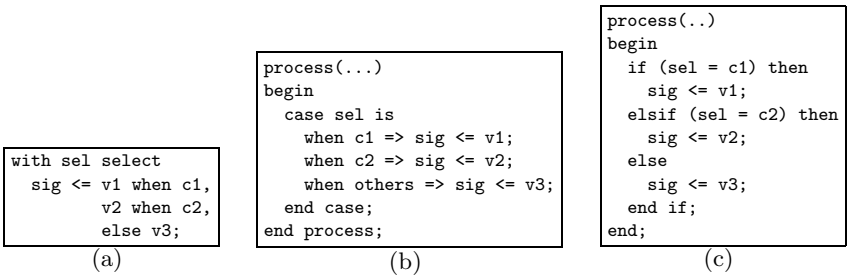

Fig. 2. A conversion of (a) *selected signal assignments* and (b) `case` statements to (c) `if` statements

Normalization of `if` Statements. After the pre-processing done above, the architecture of the component being examined is described by a set of parallel assignments and a set of `processes`, every such a process consists of a sequence of sequential assignments and (possibly nested) `if` statements. As we have already said, these sequential statements inside the processes are *not* executed sequentially—instead, for each variable, the last applicable assignment is searched and used, and all the statements preceding it are ignored. For example, for a sequence `v <= e1; if c then v <= e2; endif;`, if `c` holds, one performs the `v <= e2;` assignment, otherwise one performs the assignment `v <= e1;` (we may assume that the processes consist solely of assignments and—possibly nested—`if` statements).

In order to make dealing with the described semantics easier when generating the intermediate behavioural model, we perform one more pre-processing step. In particular, we transform each process into a single nested `if` statement in which it is clear under which conditions which assignment is to be applied (e.g., the example we mentioned above will be transformed to the statement `if c then v <= e2; else v <= e1; endif;`—more examples will come below). More precisely, for each sequential process and each variable v assigned by that process, we do the following steps (we ignore all assignments to other variables when handling v):

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

2.3 Counter Automata

For an integer arithmetic formula φ, let $FV(\varphi)$ denote the set of free variables of φ.[3] For a set of variables X, let $\Phi(X)$ denote the set of integer arithmetic formulae with free variables from $X \cup X'$ where $X' = \{x' \mid x \in X\}$. If $\nu : X \to \mathbb{Z}$ is

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

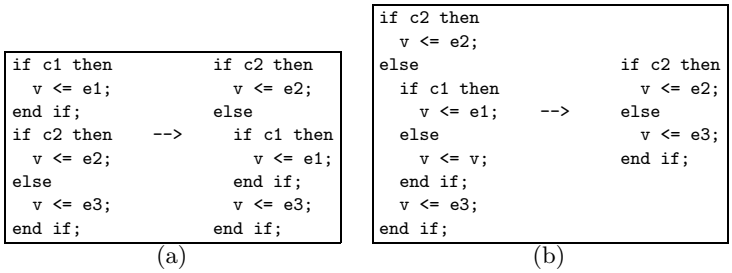

Fig. 3. Transformations of sequential statements: (a) moving all statements preceding an `if` statement to its `else` branch, (b) removing statements preceding an assignment (and thus being useless)

an assignment of $FV(\varphi) \subseteq X$, we denote by $\nu \models \varphi$ the fact that ν is a satisfying assignment of φ. A *counter automaton* (CA) is a tuple $A = \langle X, Q, q_0, \varphi_0, \rightarrow \rangle$ where X is a finite set of counters, Q is a finite set of control locations, $q_0 \in Q$ is a designated initial location, φ_0 is an arithmetic formula such that $FV(\varphi_0) \subseteq X$, describing an initial assignments of the counters, and $\rightarrow \in Q \times \Phi(X) \times Q$ is a finite set of transition rules.

A configuration of a CA is a pair $\langle q, \nu \rangle \in Q \times (X \rightarrow \mathbb{Z})$. The set of all configurations is denoted by $\mathfrak{C}$. The transition relation $\xrightarrow[A]{\varphi} \subseteq \mathfrak{C} \times \mathfrak{C}$ is defined by $(q, \nu) \xrightarrow[A]{\varphi} (q', \nu')$ iff there exists a transition $q \xrightarrow{\varphi} q'$ such that if σ is an assignment of $FV(\varphi)$, where $\sigma(x) = \nu(x)$ and $\sigma(x') = \nu'(x)$, we have that $\sigma \models \varphi$ and $\nu(x) = \nu'(x)$ for all variables x with $x' \notin FV(\varphi)$. We denote by $\xrightarrow[A]{}$ the union $\bigcup_{\varphi \in \Phi} \xrightarrow[A]{\varphi}$, and by $\xrightarrow[A]{*}$ the reflexive and transitive closure of $\xrightarrow[A]{}$. A *run* of A is a sequence of configurations $(q_0, \nu_0), (q_1, \nu_1), (q_2, \nu_2) \ldots$ such that $(q_i, \nu_i) \xrightarrow[A]{} (q_{i+1}, \nu_{i+1})$ for each $i \geq 0$ and $\nu_0 \models \varphi_0$.

2.4 Handling VHDL Integer Variables in Counter Automata

When translating operations on integer variables used in VHDL to operations on counters, we have to take care of the fact that in VHDL, arithmetical operations over integers are always implicitly evaluated *modulo the range of the appropriate integer variables*. In counter automata, we have to make the modulo computation explicit (e.g., an assignment `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 (and if we add them in the future, then also the bit-wise

manipulations on integer variables). Given a concrete counter automata analyser, the translation may need to be adjusted to respect the operations that the tool supports. If the tool does not offer all the needed operations (nor allows their implementation based on other supported operations), one has to restrict to the case when the appropriate integer variables have a fixed range (i.e. are not parameters) and can also be recorded as a part of the control states of counter automata.

3 An Intermediate Behavioural Model

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

3.1 A Definition of the Intermediate Behavioural Model

The *intermediate behavioural model* of a hardware component is defined as a triple $M = (V, T, B)$, where V is a set of variables that are typed by a function $T : V \to \{\texttt{bool}, \texttt{int}\}$, and B is a 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 $T(v) = \texttt{bool}$),
- $(v, \texttt{negedge}) \in \overline{V}$, 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 $\perp \in E$ denotes an *empty* expression (see below).

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

$$c \rightarrow v := e$$

for $c \in C^*$ being a list of enabling conditions, $v \in V$ the variable set by the rule, and $e \in E$ being an expression defining the new value of v. In other words, b

with $c = c_1c_2...c_n$ says that if $c_1 \wedge c_2 \wedge ... \wedge c_n$ holds for the evaluation of the variables, v will get a new value obtained by an evaluation of e. If $c = \varepsilon$, we consider it to be always true, and the assignment $v := e$ is always enabled.

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

3.2 Extracting Behavioural Rules from the Source Code

The architecture of a VHDL component consists of a set of parallel assignments and a set of sequential processes. With respect to the simple VHDL transformations described in Section 2.2, we may assume that the sequential processes consists of a single `if` statement for every variable set within it. In order to obtain the set of behavioural rules B from such a description, we extract the rules from VHDL statements as follows:

1. For each parallel assignment `v <= e;`, we add a rule $\varepsilon \rightarrow v := e$ into B.
2. For each sequential process that sets a variable v by a single, possibly nested, `if` statement (after the pre-processing, there is no other possibility), we proceed as follows. For each assignment statement `v <= e;` that appears on the leaf level of such a (nested) `if` statement, we add a rule $c'_1, c'_2, ..., c'_n \rightarrow v := e$ 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. 4.

3.3 Adjustments of Behavioural Rules

The Environment of a Component. To be able to model check a component, we need a model of its environment too. Currently, we model the environment to behave in a completely random way. To do that, we extend the intermediate behavioural model by adding behavioural rules for all component inputs. For every such an input $v \in V_i$, we add the following behavioural rule $\varepsilon \rightarrow v := random$. Here, *random* represents a random integer or boolean value. Note that we have to adjust the form of *random* such that the CA analyser that want to use understands it.

Non-state Variables. We are only interested in stable states that are defined by the so-called *state variables*. In the hardware developers' jargon, such variables are also known as registers or signals which save their value. The remaining variables are *non-state variables* whose values are not registered and that, from our point of view, represent just a symbolic name for some expression. From a set of behavioural rules, a non-state variable can be identified by the fact

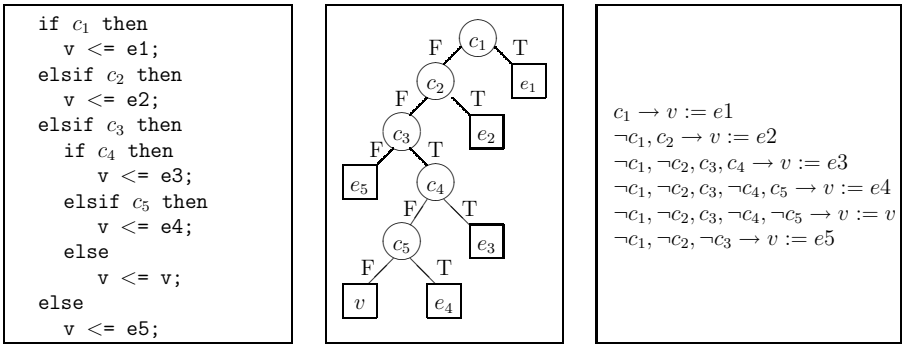

Fig. 4. Synthesis of behavioural rules wrt. the conditions passed till a certain assignment can be fired: (a) a normalized VHDL `if` statement, (b) the tree representing branching conditions, (c) the set of behavioural rules for variable v

that its value is set by a rule with the empty enabling condition (i.e. by an unconditional assignment[4]). The remaining variables are then state variables. The only exception are input variables whose values are defined and held by the environment of the modelled component. Formally, $v \in V \setminus V_i$ is a non-state variable iff $cond(b) = \varepsilon$ for the rule $b \in B$ such that $B(v) = \{b\}$. Let further $V_s = V_i \cup \{v \mid v \in V, cond(b) \neq \varepsilon\}$ be the set of state variables. Before generating counter automata, we change the intermediate behavioural model to use the state variables only. We remove the non-state variables v defined by rules $\varepsilon \rightarrow v := e$ present in B by iteratively searching for references to such variables in enabling conditions and value expressions of the rules in B and by replacing these references by e.

Behavioural Rules Over 1-bit Variables. Next, for technical reasons allowing us to ease the subsequent construction of CA 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 $b : c \rightarrow v := e$ over a 1-bit state variable $v \in V_s, T(v) = bool$, to the rule $b_{new} : c,\ v' = e \rightarrow v := e$.

Triggers of Behavioural Rules. Let $V_{\uparrow\downarrow} = \overline{V} \cap (V \times \{\texttt{posedge}, \texttt{negedge}\})$ be the set of edges of the values of variables from V. We define a mapping $R :\ B \rightarrow \{\tau\} \cup V_{\uparrow\downarrow}$ that assigns each rule either τ in case the rule models an assignment in the transparent mode or a signal edge (i.e. a *trigger*) that activates the rule if it models an assignment in the synchronous mode. Formally, for $b \in B$, let $R(b) = \tau$ iff $F(cond(c)) \cap V_{\uparrow\downarrow} = \emptyset$, and let $R(b) = t$ iff $F(cond(b)) \cap V_{\uparrow\downarrow} = \{t\}$ for some $t \in V_{\uparrow\downarrow}$. Note that this definition is correct as due to the hardware

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

description principles, there can be at most one positive or negative edge variable reference in a behavioural rule condition. Designs violating this requirement are exposed during the synthesis process.

For each rule $b \in B$ that works in the transparent mode, i.e. $R(b) = \tau$, we adjust the condition and assignment part of b such that each variable reference that appears there refers to the future. This is, we change every variable reference v that appears in $value(b)$ or $cond(b)$ to v'. The reference to the future assures that the rule is evaluated using values of variables that are computed at the same time step as the one at which we perform the evaluation (and not a step before as in the case of the synchronous mode). This is because gates working in the transparent mode immediately propagate their input values to the output. We can afford to use this transformation as we excluded the possibility of cyclic dependencies of the values of variables in the transparent mode. That is why, the variables changing in the transparent mode can be ordered according to their dependencies and evaluated in the given order starting with variables that are assigned a constant value (which happens, e.g., when the circuit is being reset) or from variables which are not changing at the given time step. For an illustration of this behaviour, see Fig. 5.

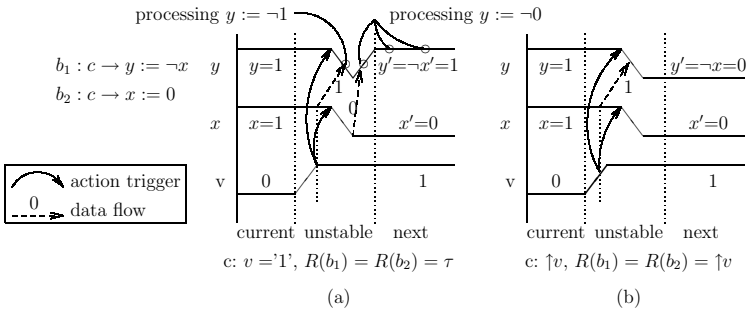

Fig. 5. A timing diagram illustrating the differences between the transparent and synchronous mode. For the transparent mode (a), both x and y are controlled by the *level* of the variable v, which causes a continuous change of their values (y is set to the negation of x via b_1, y is set to 0 via b_2). Due to the propagation delays of hardware which implements such a behaviour, there are several changes of the values until they are all stabilised, which we are, however, not interested in. The important thing to notice is that the resulting value of x is $\neg y'$ and not $\neg y$. On the other hand, in the synchronous mode (b), an *edge* triggers a change of the value of v, which holds until the next triggering event. In this case, the resulting value of x is $\neg y$ (and not $\neg y'$).

We have to do a similar adjustment as above also for the rules modelling the synchronous mode. For simplicity, we consider here the case of positive edges only. The case of negative edges is analogical. Within each rule $b \in B$ for which $R(b) = \uparrow v$ for some $v \in V$, $cond(b) = c_1 c_2 \ldots c_n \uparrow v c_{n+1} \ldots c_m$ for some $n, m \in \mathbb{N}$. Note that $F(c_1 c_2 \ldots c_n) \cap V_{\uparrow\downarrow} = \emptyset$. In this case, the way our algorithm for

generating behavioural rules works implies that the set of generated behavioural rules B must also include behavioural rules $b_\tau \in B$ whose condition is built solely of the conditions $c_1, c_2, \ldots, c_n$ (possibly negated), hence $R(b_\tau) = \tau$. Due to the evaluation order of the conditions, the b_τ rules have a priority over b. At the same time, they model the transparent mode, hence they will work with the future values of variables. That is why, in order to exclude a possible conflict of the rules b_τ with b, we have to replace every variable reference v to v' in $c_1, c_2, \ldots, c_n$ in b. Then, if some of the b_τ rules is enabled, b is disabled as its enabling condition contains a negation of some of the enabling conditions of b_τ evaluated on the same values of variables. On the other hand, if this is not the case, the rest of b will work with the current values of the variables.

4 Generating Counter Automata

4.1 Counters, Control Locations, and Initialisation

Let us fix a hardware design with a set of variables V of types T and with a set of behavioural rules B generated from the design. We start building the counter automaton A representing the design by defining its *set of counters* as all integer-type state variables from V—formally, wrt. the definition of counter automata (Def. 2.3), $X = \{v \mid v \in V_s,\ T(v) = int\}$.

Further, we build control locations of A based on all possible evaluations 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 evaluation 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 initialise the variables. The obtained values of control state variables make up the definition of the initial location q_0, the evaluation of integer variables allows us to construct the initial constraint φ_0 on counters[5]. If the modelled component has no resetting signals or the desired initial state is not the reset state, the initialisation must be defined explicitly by the user.

[5] 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.

4.2 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 evaluation 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 evaluation 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:[6]

1. For each $b \in B$ with $cond(b) = c_1 c_2 \dots c_n$ for some $n \in \mathbb{N}$, we (as far as possible) evaluate the enabling condition of b, i.e. we compute $guard^{q_1,q_2}(b) = \bigwedge_{1 \leq i \leq n} c_i^{q_1,q_2}$. Let $B_e = \{b \mid b \in B, var(b) \in V_s, guard^{q_1,q_2}(b) \neq false\}$ be the set of all (conditionally) enabled behavioural rules setting the value of state variables.
2. We further one-by-one consider all subsets $B_t \subseteq B_e$ such that B_t contains exactly one rule b such that $var(b) = v$ for each state variable $v \in V_s$. For each B_t, we perform the following steps:
   (a) In each rule $b \in B_t$, we iteratively substitute all references to the future values of counter variables by the expressions assigned to them within B_t. This is, we substitute each v' for $v \in V_s \setminus V_q$ by the expression $value(b_v)$ where $b_v \in B_t$ and $var(b_v) = v$.[7] We repeat this step till all references to future values of counters disappear.
   (b) Based on the set of rules B_t, we create a transition $q_1 \xrightarrow{\varphi} q_2$ of A where $\varphi = (\bigwedge_{b \in B_t} guard^{q_1,q_2}(b)) \wedge (\bigwedge_{b \in B_t, var(b) \notin V_q} \alpha(value^{q_1,q_2}(b))$ and α is a function that transforms an assignment $v := e$ to a formula $v' = e$.

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

Suppose now that, for instance, $V_s = \{v_1, v_2, v_3, v_4\}$ where only v_4 is a 1-bit variable, and the first step of the algorithm yields a set of rules $B_e = \{g_1 \rightarrow v_1 :=$

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

[7] At this point, only the variables representing counters are considered as the references to future values of control state variable are taken care through the partial evaluation of the expressions.

$f_1(v'_2), g_{2,1} \rightarrow v_2 := f_{2,1}(v'_3, v_1), g_{2,2} \rightarrow v_2 := f_{2,2}(v_3), g_3 \rightarrow v_3 := f_3(v_2), v'_4 = \neg v_4 \rightarrow v_4 := \neg v_4\}$ (the rule for v_4 is transformed as we described in Section 3). We can find two subsets B_t that are to be handled by the second step of the algorithm—namely, $B_{t,1} = \{g_1 \rightarrow v_1 := f_1(v'_2), g_{2,1} \rightarrow v_2 := f_{2,1}(v'_3, v_1), g_3 \rightarrow v_3 := f_3(v_2), v'_4 = \neg v_4 \rightarrow v_4 := \neg v_4\}$ and $B_{t,2} = \{g_1 \rightarrow v_1 := f_1(v'_2), g_{2,2} \rightarrow v_2 := f_{2,2}(v_3), g_3 \rightarrow v_3 := f_3(v_2), v'_4 = \neg v_4 \rightarrow v_4 := \neg v_4\}$. If we apply the steps described above for $B_{t,1}$, we obtain two CA transitions with a formula $g_1 \wedge g_{2,1} \wedge g_3 \wedge v'_1 = f_1(f_{2,1}(f_3(v_2), v_1)) \wedge v'_2 = f_{2,1}(f_3(v_2), v_1) \wedge v'_3 = f_3(v_2)$ going between control states q_1 and q_2 such that $q_1(v_4) = \neg q_2(v_4)$. Note that the condition $v'_4 = \neg v_4$ does not appear in the formula of the transition as its evaluation wrt. q_1, q_2 yields *true*.

5 Handling the Reachability Properties to Be Verified

In our work, we concentrate on verifying that certain *bad configurations* are not reachable. We assume the bad configurations to be given by a boolean VHDL expression—an *error condition*. The error condition may refer to 1-bit VHDL variables appearing in the design of the component being checked (which are represented as a part of the control location of the generated counter automata) as well to VHDL bit-vector variables (represented by the values of counters in the counter automata).

In order to facilitate verification of reachability of the bad configurations, we extend a generated counter automaton by a special *error state* whose reachability implies that a bad configuration is reachable in the component being checked. The error state is connected to the control states of the generated counter automaton that represent a valuation of the VHDL 1-bit variables which is not contradictory with the error condition. Moreover, the transitions to the error state are guarded by conditions on counters derived from the error condition by substituting the 1-bit variables by values that appear in the source control location of these transitions (after which, just a constraint on bit-vector variables remains).

6 Experiments

For our experiments, we implemented a Python-based prototype [16] of the proposed translation (up to some of the issues of the VHDL pre-processing mentioned in Section 2). In particular, we implemented a translation to counter automata in the input language of the ARMC tool [15] and also to integer programs in the C programming language in order to be able to use the Blast model checker [8] as well. Both of the tools provide us with the possibility of verifying reachability properties of counter automata (or, alternatively, integer programs) using techniques based on predicate abstraction and the counterexample-guided abstraction refinement (CEGAR) loop.

To test the proposed counter-automata-based model extraction method, we have first applied it to two small non-parametric components (having integer

Table 1. Experiments with counter automata extraction from VHDL and with their subsequent reachability analysis using ARMC and Blast

Component	Locations	Transitions	Counters	Extraction time	ARMC	Blast
Counter	5	13	2	< 1s	< 1s	1.5s
Register	9	43	2	1s	< 1s	< 1s
Synchronous LIFO	65	985	3	24s	40s	5m31
Asyn. FIFO (FE)	65	5060	12	1m12s	6m56s	N/A
Asyn. FIFO (Status)	129	6628	12	4m	4m16	N/A

variables, but of a fixed width). Then we applied the method to two more complex parametric components, including a real-life, highly specialised, parametric component developed within the Liberouter project [13].

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 work 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 parametrisation of the number of items the LIFO can save. This component implements—among other—signals that say whether it is empty or full. We verified whether these signals are always correctly set for any possible size of the LIFO.

The last verified component is an *asynchronous queue* (FIFO). This specialised parametric component was built to be used in network monitoring adaptors developed within the Liberouter project (with a stress on being as efficient as possible). Apart from signals about whether the component is empty or full, it also implements additional signals saying whether it is almost full or almost empty (less than some amount of items are free/occupied). For the component, we successfully verified two properties: (i) that the queue does not inform that it is empty and full at the same time, and (ii) that the status information about the queue being almost full is set correctly. For a more detailed description of the verified properties see [16].

The results of our experiments are summarised in Table 1. The first column gives the verified component—for the last component, there are two lines corresponding to the two different properties that we checked for it. The next column provides the number of control locations in the generated counter automata—note that the number corresponds to $2^n + 1$, which is the number of control locations over n 1-bit state variables, plus one location representing the bad state. The next two columns provide the number of transitions between control locations of the generated counter automata and the number of used counters (integer variables). The next columns gives the times used by our prototype tool to generate the counter automata. Finally, the last two columns provide the time used by ARMC and Blast, respectively, to verify the generated counter

automata. The experiments were performed on an Intel Xeon X5355 processor with 16GB of memory. ("N/A" means that the verification did not finish.)

7 Conclusion and Future Work

We have presented a new, quite general and automated, approach to formal verification of parametrised VHDL components. The approach is based on an automated translation of the components to counter automata and on exploiting the constantly improving technology for verifying counter automata (or integer programs). We have built a prototype tool implementing our translation schema and successfully used it together with the ARMC tool [15] for verification of several interesting properties of parametrised VHDL components, including a real-life component developed within the Liberouter project [13].

In the future, we want to experiment with lifting some of the restrictions of our initial approach (e.g., allowing a bit-wise approach to parametrised components). Another interesting research direction is to investigate possibilities of reducing the size of the automata that we generate. Further, we would like to do more experiments with real-life components and also with using more different tools for handling counter automata (or integer programs).

Acknowledgement. We would like to thank Andrey Rybalchenko for his help with the use of the ARMC tool.

References

1. Bardin, S., Finkel, A., Leroux, J., Petrucci, L.: FAST: Fast Acceleration of Symbolic Transition systems. In: Hunt Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, Springer, Heidelberg (2003)
2. Bardin, S., Finkel, A., Lozes, E.: From Pointer Systems to Counter Systems Using Shape Analysis. In: Proc. of AVIS 2006 (2006)
3. Bouajjani, A., Bozga, M., Habermehl, P., Iosif, R., Moro, P., Vojnar, T.: Programs with Lists are Counter Automata. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, Springer, Heidelberg (2006)
4. Chaki, S., Clarke, E., Groce, A., Ouaknine, J., Strichman, O., Yorav, K.: Efficient Verification of Sequential and Concurrent C Programs. Formal Methods in System Design 25(2–3), 129–166 (2004)
5. Chu, P.P.: RTL Hardware Design Using VHDL: Coding for Efficiency, Portability, and Scalability. John Wiley and Sons, Inc, Hoboken, New Jersey (2006)
6. Comon, H., Jurski, Y.: Multiple Counters Automata, Safety Analysis and Presburger Arithmetic. In: Vardi, M.Y. (ed.) CAV 1998. LNCS, vol. 1427, Springer, Heidelberg (1998)
7. Habermehl, P., Iosif, R., Rogalewicz, A., Vojnar, T.: Proving Termination of Tree Manipulating Programs. Verimag, TR-2007-1 (2007), `www-verimag.imag.fr/index.php?page=techrep-list`
8. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Software Verification with Blast. In: Ball, T., Rajamani, S.K. (eds.) Model Checking Software. LNCS, vol. 2648, Springer, Heidelberg (2003)

9. Kořenek, J., Pečenka, T., Žádník, M.: NetFlow Probe Intended for High-Speed Networks. In: Proc. of FPL 2005, IEEE Computer Society, Los Alamitos (2005)
10. Minsky, M.L.: Computation: Finite and Infinite Machines. Prentice-Hall International, Englewood Cliffs (1967)
11. IEEE Computer Society. IEEE Std 1076-2000. IEEE Standard VHDL Language Reference Manual. IEEE Std 1076-2000. Pages. 290. (2000) ISBN: 0-7381-1948-2
12. Leonardo Spectrum, Mentor Graphics (2007), `www.mentor.com/products/fpga_pld/synthesis/leonardo_spectrum`
13. Liberouter Project Homepage. `www.liberouter.org`
14. ModelSim, Mentor Graphics (2007), `www.model.com`
15. Podelski, A., Rybalchenko, A.: ARMC: The Logical Choice for Software Model Checking with Abstraction Refinement. In: Hanus, M. (ed.) PADL 2007. LNCS, vol. 4354, Springer, Heidelberg (2006)
16. Smrčka, A.: VHD2CA. In: A Prototype of a Translator from VHDL to Counter Automata, `www.fit.vutbr.cz/~smrcka/projects/vhd2ca/`
17. Šafránek, D., Smrčka, A., Vojnar, T., Řehák, V., Řehák, Z., Matoušek, P.: Verifying VHDL Design with Multiple Clocks in SMV. In: Brim, L., Haverkort, B., Leucker, M., van de Pol, J. (eds.) FMICS 2006 and PDMC 2006. LNCS, vol. 4346, Springer, Heidelberg (2007)
18. Yavuz-Kahveci, T., Bartzis, C., Bultan, T.: Action Language Verifier, Extended. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, Springer, Heidelberg (2005)