Checking and Deriving Module Paths in Verilog Cell Library Descriptions

Matthias Raffelsieper
CS Dept., TU/Eindhoven
Eindhoven, The Netherlands
Email: M.Raffelsieper@tue.nl

MohammadReza Mousavi
CS Dept., TU/Eindhoven
Eindhoven, The Netherlands
Email: M.R.Mousavi@tue.nl

Chris Strolenberg
Fenix,
Eindhoven, The Netherlands
Email: chris@fenix-da.com

***Abstract*—Module paths are often used to specify the delays of cells in a Verilog cell library description, which define the propagation delay for an event from an input to an output. Specifying such paths manually is an error prone task; a forgotten path is interpreted as a zero delay, which can cause further flaws in the subsequent design steps. Moreover, one can specify superfluous module paths, i.e., module paths that can never occur in any practical run of the model and hence, make excessive restrictions on the subsequent design decision. This paper presents a method to check whether the given module paths are reflected in the functional implementation. Complementing this check, we also present a method to derive module paths from a functional description of a cell.**

I. Introduction

A *cell library* is a collection of logic cores used to construct larger chip designs, consisting of combinational cells (e.g., **and** and **xor**) and sequential cells (e.g., latches and flip-flops). Cell libraries are usually described at multiple levels of abstraction, such as a transistor implementation that will finally be manufactured and a Verilog description that is used for simulation. In [8] a formal semantics for the VeriCell subset of Verilog was provided, which was used as a basis for formal conformance checking between the two levels of abstraction. The VeriCell subset of Verilog is the subset commonly used for the functional description of cell libraries. However, Verilog descriptions of cells also contain timing specifications, in terms of *module path delays* (also called *timing arcs*, *delay arcs*) and *timing checks*.

Using symbol execution for detecting paths has been studied in various domains, see e.g., [3], [2]. The main goal of these studies has been to make the calculation of delays or worst case execution times more precise. For example, in [3] an accurate timing analysis for combinational circuits is introduced and in [2] symbolic simulation is used to verify the timing of FSM models of sequential circuits. Since the goal of the work reported in [2] is different, some extra assumptions on the structure of circuits had to be made in [2] to make the timing verification feasible. Elsewhere, in [7], an analysis technique was presented to detect missing timing checks. In this paper, we focus on analyzing module path delays. We present techniques to check whether a given module path is consistent with the functional description, and to derive module paths from the functional description. The actual delays put on module paths are irrelevant for our analysis since at this level, we are only concerned with their necessity and realizability, i.e., the need and use of their presence or absence. This forms the basis for any subsequent timing analysis, i.e., one has to first determine whether a module path can actually occur or not before one can determine the delays specified by module paths. Furthermore, when considering all of the specified paths of a cell, one might get *false paths* during the timing closure of a larger design composed of multiple cells, which might reject correct designs. Therefore, we check whether the given module paths are consistent with the functional description.

Complementing the check, we also present a technique to derive module paths from a functional description. This allows to guarantee that all possible delay behaviors of a cell have been considered. If some module path would be forgotten, then simulators will treat this as a zero delay, i.e., the output will change instantly. This can cause further flaws in the subsequent design steps and the timing simulation of such a circuit does not reflect the actual behavior.

The rest of this paper is structured as follows. In Section II we introduce the VeriCell subset of Verilog that we are interested in. Furthermore, this section also shortly explains our representation of transition systems, which is used to represent the semantics of VeriCell defined in [8]. Section III then defines the formal requirements imposed by a module path onto the functional description. These requirements are used to check whether the module paths specified for a cell are consistent with its functional description, thereby finding module paths that are superfluous and can be removed. For the other direction, namely searching for missing module paths, a method is presented in Section IV. There, the functional description of a cell is analyzed and changes of an output as a result of a changing input are extracted. Together with some additional analysis of the output value, we thereby derive all possible module paths from the functional description of a cell. Our research is driven by a cooperation with Fenix as industrial partner. Hence, we applied both methods to industrial cell libraries and present some of our experimental results in Section V. We conclude the paper in Section VI.

II. Preliminaries

In this paper, we are concerned with a subset of Verilog, called VeriCell in [8], which is the subset used in the functional description of cell libraries. Hence, in this section, we introduce this subset. In Verilog, variables can take the values 0, 1, X, and Z, corresponding to the Boolean values false and true, an unknown value, and a high impedance, respectively. However, in VERICELL, the values X and Z behave equivalently. Hence, our domain of interest are the *ternary values* $\mathbb{T} = \{0, 1, \mathsf{X}\}$.

A *cell* is a module that contains only instantiations of built-in and user defined primitives. Furthermore, it may contain a timing specification, specifying propagation delays for events. We illustrate the syntax of VERICELL by means of the following example.

Consider the cell depicted in Figure 1, which is the functional description of a D flip-flop with active low reset and the module paths of its timing specification. This cell is constructed from two instances of the built-in primitives **buf** and **not**. Furthermore, it contains two instances of the User Defined Primitive (UDP) latch, whose definition is given at the top. This UDP implements a storage latch that is transparent when its ck input is 1 and that stores its current value if the ck input is 0. Furthermore, it can be reset by setting input rb to 0. This behavior is implemented in the table given in the UDP. The UDP is evaluated by looking up a row that matches the (previous and the current) input values and the previous value of the output and then copying the defined value to the output (from left to right, the values before the first colon are those of inputs, the one between the two colons is the previous value of the output and the one after the second colon is the prescribed next value of the output). Entries 0, 1, and x match exactly their corresponding values, whereas ? matches any value. Entries 0, 1, x and ? are called *levels*. Next to levels, one can also give *edge* specifications, which not only match the current value of the input, but transitions of an input. Entry * in the table specifies an edge specification, which matches a transition of an input from any value to any other value. Finally, the last column may, besides the specifications 0, 1, and x, also contain the entry -. This represents a non-changing output, i.e., if such a row matches, the old output value is also the new output value. In case no row matches in a UDP, then the new value of the output is defined to be X. A formal definition of evaluating UDPs and built-in primitives is given in [8].

In the timing specification of our example cell, there are two module paths which are both *edge-sensitive* paths, i.e., they are active whenever a certain change in an input is exhibited. The first module path describes that when the reset has a falling edge (i.e., an edge towards 0), then the output q changes its value after t_rst time units. The *data source expression* `+: 0` describes that the output will change its value to 0. Similarly, the second module path defines that on a positive edge of the clock, the output q will change its value to the value of d. However, this module path is a *state-dependent* module path, since it only applies when the condition rb == 1 is true, as specified in **if** preceding the module path.

We interpret such a cell with k inputs $i_1, \ldots, i_k$, m outputs $q_1, \ldots, q_m$, and n sequential variables $s_1, \ldots, s_n$ as Boolean equations, using the semantics presented in [8]. Here, we let I denote the space of all possible inputs $I = \mathbb{T}^k$, O denote all possible outputs $O = \mathbb{T}^m$, and S denote all possible states $S = \mathbb{T}^{n+k}$, in which we also remember the previous values of the inputs, denoted $i_1^p, \ldots, i_n^p$, to detect transitions. For $l \in \mathbb{N}$ and a vector $\vec{v} = (v_1, \ldots, v_l) \in \mathbb{T}^l$, we let $\vec{v}|_j = v_j$ denote the j-th component of that vector, for all $1 \leq j \leq l$. This definition is lifted to sets of vectors by defining $V|_l = \bigcup_{\vec{v} \in V} \{\vec{v}|_l\}$ for $V \subseteq \mathbb{T}^l$. Furthermore, we define the *concatenation* of vectors $\vec{u} = (u_1, \ldots, u_a) \in \mathbb{T}^a$ and $\vec{v} = (v_1, \ldots, v_b) \in \mathbb{T}^b$ as $\vec{u} +\!\!+ \vec{v} = (u_1, \ldots, u_a, v_1, \ldots, v_b) \in \mathbb{T}^{a+b}$. Given a vector $\vec{u} = (u_1, \ldots, u_l) \in \mathbb{T}^l$, a value $v \in \mathbb{T}$, and some $1 \leq j \leq l$ we define the *substitution* of the j-th component of $\vec{u}$ with the value v as $\vec{u}[j := v] = (u_1, \ldots, u_{j-1}, v, u_{j+1}, \ldots, u_l)$.

```
primitive latch(q, d, ck, rb);
  output q; reg q; input ck, d, rb;
  table
    // d  ck  rb  : q : q'
       *   0   ?  : ? : - ;
       ?   0   1  : ? : - ;
       0   1   ?  : ? : 0 ;
       1   1   1  : ? : 1 ;
       ?   ?   0  : ? : 0 ;
  endtable
endprimitive

module dff(q, ck, d, rb);
  output q; input ck, d, rb;
  buf(q, qint);
  latch(qint, iq, ck, rb);
  latch(iq, d, ckb, rb);
  not(ckb, ck);

  specify
    (negedge rb => (q +: 0)) = t_rst;
    if (rb==1) (posedge ck => (q +: d)) = t_ck;
  endspecify
endmodule
```

Fig. 1. Verilog Source of a Resettable D Flip-Flop

Because VERICELL programs might be non-deterministic as observed in [7], we represent the computation of next states by the function $\delta : I \times S \to \mathcal{P}(S)$, where $\mathcal{P}(S)$ denotes the power set of states, computing for a given input vector $\vec{i} \in I$ and state $\vec{s} +\!\!+ \vec{i^p} \in S$ the set of all possible next states $\delta(\vec{i}, \vec{s} +\!\!+ \vec{i^p})$. Given a state $\vec{s} +\!\!+ \vec{i^p} \in S$, the function $\lambda : S \to O$ computes the values $\lambda(\vec{s} +\!\!+ \vec{i^p})$ of the outputs in that state. We extend δ and λ to sets of states in the natural way: $\delta(\vec{i}, S') = \bigcup_{s' \in S'} \delta(\vec{i}, s')$ and $\lambda(S') = \bigcup_{s' \in S'} \lambda(s')$ for each $S' \subseteq S$. As stated in the Verilog standard [5], the initial state s_0 of a cell is the state in which all variables have the value X. A state $s \in S$ is called *reachable* if there exist states $s_1, \ldots, s_n \in S$ and inputs $\vec{i}_0, \ldots, \vec{i}_{n-1} \in I$ such that $s_n = s$ and for all $0 \leq j < n$ we have $\delta(\vec{i}_j, s_j) \ni s_{j+1}$. Furthermore, we define a constrained evaluation of δ for an input vector $\vec{i} \in I$, state $s \in S$, and constraint values $\vec{c} \in \mathbb{T}^k$ as $\delta(\vec{i} \wedge \vec{c}, s) = \delta(\vec{i}, s)$ if for all $1 \leq j \leq k$ either $i_j = c_j$ or $c_j = \mathsf{X}$. Otherwise, $\delta(\vec{i} \wedge \vec{c}, s) = \emptyset$.

In the following, we interpret an *edge specification* edge $\in \{$**posedge**$,$ **negedge**$,$ `""`$\}$ as the set of transitions that it matches, where **posedge** $= \{(0, 1), (0, \mathsf{X}), (\mathsf{X}, 1)\}$, **negedge** $= \{(1, 0), (\mathsf{X}, 0), (1, \mathsf{X})\}$, and furthermore `""` $=$ `*` $= \{(u, v) \mid u \neq v\} =$ **posedge** $\cup$ **negedge**.

III. Checking Module Paths

A *module path* describes a delay between an input and an output of the cell. There are two basic types of module paths: *simple paths* and *edge-sensitive paths*. An example of a simple path is `(ck => q) = 10`, expressing that a change of input `ck` influences output `q` after a delay of 10 time units. An example of an edge-sensitive path is `(`**`posedge`**` ck => (q +: d)) = 12`, expressing that a positive edge of input `ck` affects the output `q`, which takes the value `d` (`+` indicates that the value of `d` is passed in non-inverted form), after a delay of 12 time units.

Basic paths can be used to construct the *state-dependent module paths*, which are module paths equipped with a condition. An example of a state-dependent module path is **`if`**` (rb == 1) (`**`posedge`**` ck => (q +: d)) = 12`, which specifies the same delay as the previous edge-sensitive path example, but this delay only occurs if the condition `rb == 1` holds. A condition is defined to hold if it does not evaluate to 0, i.e., if it evaluates to either 1 or X.

We will only consider state-dependent module paths in the following, since the others can be expressed as such paths by simply adding "**`if`** `(`i`==X)`" for some input i, as a comparison with the value X will always make the equality evaluate to X.

A. Requirements for Simple Module Paths

A simple module path, as its name indicates, is the simplest form of specifying that an input influences the value of an output. However, it is desirable that such an effect does actually take place. For example, one could add the simple module path `(d => q) = 1;` to the example in Figure 1, stating that a change of input `d` affects the value of output `q` one time unit after the change of input `d`. However, the output of a flip-flop will never change as a result of only changing the data input, for the output to change a positive edge of the clock is required. Hence this is a module path that never occurs in practice, which might result in too severe restrictions in the timing analysis such that a circuit using this flip-flop with the above module path would fail to meet its timing requirements, although an implementation would never suffer from this problem.

We aim to give a formal definition of the requirements for a module path to be *consistent* with the functional description. Let **`if`** `( (`i_1`==`c_1`) &&`$\ldots$`&& (`i_k`==`c_k`) ) (`i_j p`=>` q_l`)` be a state-dependent simple module path, where p is called the *polarity* of the module path, and $p \in \mathrm{Pol} = \{$`+`$,$ `-`$,$ `""`$\}$ (with `""` representing the empty string). The polarity expresses the direction of the output change: For positive polarity ($p =$ `+`) if the output changes, then the change is into the same direction as the input. For negative polarity ($p =$ `-`) the output changes into the opposite direction of the input, if it does change. For no polarity ($p =$ `""`) the output is free to change into any direction. Note that any (state-dependent) simple module path can be written in the above format, by inserting X for c_j when input i_j is not constrained in the original module path.

We define the semantics of module paths as imposing two constraints on the transition system of cells. First, we require that a state can be reached from which the specified output will change as a result of only changing the specified input. If this was not the case, then the output would never change as a result of the changing input, hence this module path would never be active and could be removed. The second constraint deals with the polarity: If it is either `+` or `-`, then we also require that in case the output changes, it changes into the direction specified by the polarity. Formally, we express these two constraints as follows:

1) There exists a reachable state $\vec{s}{+\!\!+}\vec{i^p}$, a value $v \in \mathbb{T}$, and an output value $\lambda' \in \lambda(\delta(\vec{i^p}[j := v] \wedge \vec{c}, \vec{s}{+\!\!+}\vec{i^p}))|_l$ such that $\lambda' \neq \lambda(\vec{s}{+\!\!+}\vec{i^p})|_l$.
2) If $p \in \{$`+`$,$`-`$\}$, then for all reachable states $\vec{s}{+\!\!+}\vec{i^p}$, all values $v \in \mathbb{T}$, $\lambda = \lambda(\vec{s}{+\!\!+}\vec{i^p})|_l$, and every $\lambda' \in \lambda(\delta(\vec{i^p}[j := v] \wedge \vec{c}, \vec{s}{+\!\!+}\vec{i^p}))|_l$ the following holds:
   - If $p =$ `+` then
     - If $(i^p_j, v) \in$ **posedge**, then $\lambda = \lambda'$ or $(\lambda, \lambda') \in$ **posedge**
     - If $(i^p_j, v) \in$ **negedge**, then $\lambda = \lambda'$ or $(\lambda, \lambda') \in$ **negedge**
   - If $p =$ `-` then
     - If $(i^p_j, v) \in$ **posedge**, then $\lambda = \lambda'$ or $(\lambda, \lambda') \in$ **negedge**
     - If $(i^p_j, v) \in$ **negedge**, then $\lambda = \lambda'$ or $(\lambda, \lambda') \in$ **posedge**

B. Requirements for Edge-Sensitive Module Paths

An edge-sensitive module path specifies that a certain change of the input influences the output. Hence, we again require that this effect on the output does exist when the input exhibits one of the specified transitions, like for simple module paths. Furthermore, an edge-sensitive module path also specifies the value of the output after the change by means of the data source expression. It should also be verified that this expression reflects the actual computation of the output value in the functional description.

When given a state-dependent edge-sensitive module path **`if`** `((`i_1`==`c_1`) &&`$\ldots$`&& (`i_k`==`c_k`)) (edge` i_j `=> (`q_l p`:` d`))` with polarity $p \in \mathrm{Pol}$ and edge $\in \{$**posedge**$,$ **negedge**$,$ `""`$\}$, we again have two constraints to be satisfied. The first one is the same as constraint (1) for simple module paths, except that we now restrict the input change to be one of the transitions contained in the specified edge. The second constraint deals with the polarity and the data source expression specified in an edge-sensitive module path. We require that when the polarity is either positive or negative, respectively, then the output has the value of the data source expression or its negation (depending on the polarity) whenever the input makes one of the specified transitions. Hence, we require that the module path is realizable in some reachable state and that its data source expression corresponds to the output value in all reachable states triggering the module path.

Formally, we require constraint (1) to hold for one of the given input transitions and furthermore, if $p \in \{+, -\}$, we require for all reachable states $\vec{s}+\!\!+\vec{i^p}$ and values $v \in \mathbb{T}$ with edge $\ni (i_j^p\ v)$ that $\lambda(\delta(\vec{i^p}[j := v] \wedge \vec{c}, \vec{s}+\!\!+\vec{i^p}))|_l = \{q_l'\}$, where $q_l' = d$ if $p = +$ and $q_l' = \neg d$ if $p = -$.

C. Reachability Checking of Module Paths

Using the formal requirements of module paths as given above, one can check them in the functional implementation of the circuit. For this purpose, we use the semantics developed in [8] to get a symbolic representation of the functional description of a cell. By definition, all variables in Verilog have initially the value X. Starting from this state, we then search for a reachable state that satisfies condition (1) using our implementation of a symbolic simulator. If no such state exists, then the input of the module path never affects the output, hence this module path is infeasible and can be removed. Otherwise, we perform another reachability check. This time, we check whether from the initial state another state can be reached in which a counterexample to the specified behavior occurs. For simple module paths, this occurs when a polarity is specified, but the output changes in the opposite direction. For an edge-sensitive module path with specified polarity, we check that the data source expression always determines the value of the output after the specified input transition of the input. For this purpose, we search for a reachable state in which one of the given input transitions has been applied, but the output does not have the value given by the data source expression. If such a state is reachable, then we have found a counterexample to the module path and output it as incorrect. Otherwise, for all reachable states the data source expression represents the value of the output.

However, with the above notion of inconsistent module paths, usually only a very small number of module paths are feasible. This is usually due to the value X. Since digital circuits only operate on two logical values (the Boolean values false and true), this value is commonly interpreted as unknown in the two-valued logic. For example, a transition from 0 to X could either be a transition from 0 to 1 or no transition at all. In order to cope with practical examples we allow to strengthen the module paths by deviating from the behavior defined in the Verilog standard [5], which defines X to be a third value unrelated to both 0 and 1 (see [9] for an in-depth discussion of the problems associated with the value X). If for example we have an edge specification **posedge**, then we only consider the binary transition $(0, 1)$ that is contained in the set of transitions it matches. This amounts to viewing X as representing either 0 or 1, since $(0, 0)$ is not an edge we do not have to consider it. Furthermore, we also allow to strengthen the conditions: Previously, and as required by the Verilog standard [5], a condition was active if it either evaluates to 1 or to X. Now we require both sides of an equation to be equal. For comparisons with Boolean values, this again amounts to viewing X as either 0 or 1. These two strengthenings can be enabled by options passed to our implementation. It will then check the conditions as described above for the strengthened module paths. If they are consistent with the functional description, then they faithfully specify the behavior of the functional description, under the condition that the strengthenings hold.

For our example cell in Figure 1, the non-strengthened module paths are not consistent with the functional description. The first module path (**negedge** rb => (q +: 0)) specifies that the output q changes to the value 0 if the input rb exhibits one of the transitions that are considered a **negedge**. When looking at the table of the UDP latch in Figure 1, then one can observe that the output is set to 0 when input rb is 0. However, as defined in Section II, we also have the transition $(1, \mathsf{X}) \in$ **negedge**, i.e., the value of input rb after this transition is X. For this transition, the value of output q will be set to X, since none of the rows of UDP latch match in this case. Similarly for the second module path: The data value is only latched if the clock has value 1, however the **posedge** allows it to change from 0 to X. Furthermore, for this module path the condition rb == 1 is true if input rb is either 1 or X, since X == v is true for all values v. If however the input rb has the value X, then the data input will not be latched since it is unclear whether the reset is active or not. Hence, in this case the data source expression does not always correspond to the value of the output q, even when restricting to binary values for input ck. If we also strengthen the condition, and require that both sides of it have equal values, then this module path is only active when the reset signal rb has value 1. In this case, the data source expression does express the value of the output q, as intended.

IV. Deriving Module Paths

When given a cell library, all possible module paths have to be specified, otherwise the simulation will use no delay and therefore might not faithfully model the real implementation. Hence, a method is desirable to automatically extract all possible module paths from a functional description.

To determine module paths for a given cell, we construct a set of Boolean equations using the semantics for VeriCell given in [8]. This gives us, for each signal in the cell, an equation computing its next output value given the current values of the inputs to the primitive driving the wire. Such a primitive might either be a built-in primitive or a combinational UDP, which then results in a combinational equation, or a sequential UDP, for which we also consider the current output value as an input, i.e., we model this as a feedback loop.

Using these equations, we then consider each triple of **posedge** or **negedge**, input, and output of the given cell. For such a triple, we construct a formula describing those states for which the specified input makes a transition matched by the edge specification and for which the output changes its value. Furthermore, the formula requires the remaining inputs to remain unchanged, since we want to determine the influence of a single input on the output.

Formally, for a triple (edge, j, l) we check reachability of a state $\vec{s} +\!\!+ \vec{i}^p$ such that there exist $v \in \mathbb{T}$, $(i^p_j, v) \in \mathsf{edge}$, and $\lambda' \in \lambda(\delta(\vec{i^p}[j := v], \vec{s} +\!\!+ \vec{i}^p))|_l$ with $\lambda' \neq \lambda(\vec{s} +\!\!+ \vec{i}^p)|_l$. This reachability check is performed using a symbolic simulator. If no such state is reachable, then the considered input transitions never have an effect on the currently considered output, and hence there should not be a module path. Otherwise, if we find such a reachable state, then the considered transition of the input can change the output value and therefore a module path should exist for this configuration. We then perform a symbolic simulation of the specified transitions of the edge for the currently considered output, using the Boolean equations. This approximates the data source expression for the module path. It is only an approximation, since we only simulate two input vectors: One where the currently considered input has the previous value of its transition and one where the currently considered input has the new value after the transition. Such a transition might occur at any time, therefore we allow the state in which the symbolic simulation starts to be arbitrary. This however includes unreachable states, i.e., states which are never reached during the operation of the cell. We simplify the data source expression by removing unreachable states. For this purpose, we convert the found data source expression into its sum-of-products representation and check every product (representing a state) for reachability. If one of these states is unreachable, then it can never contribute to the value of the output, and hence it can be removed. From the reachable products we finally construct our data source expression and print out the module path.

V. Experimental Results

We have applied the presented methods for checking module paths against and for deriving module paths from functional descriptions to cells from the Nangate Open Cell Library [6] and from proprietary cell libraries provided by our industrial partner Fenix. In the following, we mainly present our results for the Nangate Open Cell Library, since it is publicly available, and we only briefly comment on the observations made for proprietary cells. We had to leave out the cells `TBUF`, `TNOT`, and `TLAT`, since these distinguish the fourth value `Z` that is currently not supported in our semantics of VeriCell.

A. Checking Module Paths

The results of checking the module paths contained in the Nangate Open Cell Library are shown in Table I, where we only present the results for the sequential cells. For the combinational cells, all module paths were directly realizable. In columns "Cell" and "# MPs" we give the name and number of module paths in that cell, respectively; "Direct" specifies how many module paths hold directly without any strengthening, "Edge" is the number of consistent module paths when strengthening the edge, "Equiv" is the number of consistent module paths when strengthening the condition, and "Equiv & Edge" is the number of consistent module paths when strengthening both the condition and the edge. Note that in general after strengthening either the edge or the condition, module

TABLE I
Checking module paths in the Nangate Open Cell Library

Cell	# MPs	Direct	Edge	Cond	Cond & Edge	Time [s] no-pp	Time [s] pp
DFF	2	0	2	0	2	0.49	0.20
DFFR	6	0	4	0	6	6.06	0.24
DFFS	6	0	4	0	6	2.36	0.24
DFFRS	14	4	6	4	14	75.48	0.41
DLH	2	1	2	1	2	0.69	0.20
DLL	2	1	2	1	2	0.48	0.20
SDFF	2	0	0	0	2	28.96	0.21
SDFFR	6	0	4	0	6	43.18	0.35
SDFFS	6	0	4	0	6	115.50	0.32
SDFFRS	14	4	6	4	14	438.23	0.87

paths that were consistent directly are still consistent. Also, module paths that were consistent after strengthening either the edge or the condition are still consistent after strengthening both. Finally, in the last two columns we give the time it took to check all module paths. The column "Time no-pp" gives the time taken without preprocessing the model, whereas column "Time pp" gives the time required for preprocessing and checking the module paths. The preprocessing results in a symbolic next-state function δ such as considered in this paper, whereas without preprocessing we have to consider multiple smaller steps to reach a next stable state.

We tried the strengthenings in the order of the columns, i.e., if we found a module path to be inconsistent with the functional description, we first replaced the edge by an edge that does not contain any X values, and if this module path was still inconsistent we strengthened the condition, first with the general edge specification, then with the strengthened edge.

As it can be seen in the results, none of the module paths in the library were found to be inconsistent with the functional description of the cell when both strengthenings are used, since the numbers in the column "Cond & Edge" are the same as the total number of module paths in column "# MPs". Furthermore, it was never sufficient to only strengthen the condition, module paths that still were inconsistent after strengthening the edge required strengthening of the condition and of the edge. This can be seen in the results, as the numbers in the column "Direct" are always equal to the numbers in the column "Cond". We want to stress again that these strengthenings reflect a misconception between the semantics implied by the Verilog standard and the common perception of Verilog designers. Finally, we observe that the time it took to check all module paths in a cell is negligible when using the preprocessing, even when including the time preprocessing takes, as shown in the last column of Table I. Hence, this check can easily be done before doing timing analysis of larger circuits using such a cell library. When not preprocessing the cells, then the time required to check the module paths is sometimes much larger, as shown in the penultimate column of Table I.

For cells from proprietary cell libraries, we also observed that the time taken to check all module paths was usually small. Only for 2 cells, each having 224 module paths, checking the module paths with enabled preprocessing took up to 19 seconds. However, for the proprietary cells quite a number of module paths were inconsistent with the functional description, even after strengthening. For example, in a cell containing a scanable flip-flop, the scan logic was forgotten in the data source expression of a module path, i.e., the scan-enable input was assumed to be always `0`. Another reason for module path inconsistency was the assumption that all inputs to a cell are binary, i.e., either `0` or `1`. An example of this is the module path (**posedge** `CK => (Q +: !SE&D | SE&SI)`), which was specified for another scanable flip-flop with clock input `CK`, data input `D`, scan-enable `SE` and scan-input `SI`. The idea is that when `SE` is `0` then the value of `D` will become visible at the output `Q`, whereas the output value will be the value of `SI` if the input `SE` is `1`. However, in this case a non-pessimistic multiplexer was used to select between `D` and `SI`. This multiplexer also outputs `1` when both `D` and `SI` are `1`, even when `SE` is `X`. Then however, the data source expression does not describe the behavior of the circuit, since it evaluates to `X` whereas the flip-flop outputs a `1`. Hence, such a module path is only consistent with the functional description if one strengthens even more to allow only binary input values, which can also be done in our implementation. Another possibility to make the module path consistent is the description of this non-pessimistic behavior of the multiplexer, by using the data source expression `!SE&D | SE&SI | D&SI`.

B. Deriving Module Paths

We applied our method for the derivation of module paths to the Nangate Open Cell Library and to proprietary cell libraries provided by our industrial partner Fenix. We restricted the inputs to be binary, i.e., either `0` or `1`. For the combinational cells, it found the expected dependency of all inputs on the outputs. More interesting are the results for the sequential cells, for which we give as a representative the output for the cell `SDFFRS` from the Nangate Open Cell Library, describing a scanable flip-flop with active-low reset `RN` and active-low set `SN` (where we modified the data source expressions slightly to be more readable):

```
(posedge CK => (Q +: RN & (!SN | !SE&D
                          | SE&SI)   ));
(posedge CK => (QN +: SN & (!RN | !SE&!D
                           | SE&!SI)   ));
(negedge RN => (Q +: 0));
(posedge RN => (Q +: !SN));
(negedge RN => (QN +: SN));
(negedge SN => (Q +: RN));
(negedge SN => (QN +: 0));
(posedge SN => (QN +: !RN));
```

The output of our method gives a number of module paths that were found for the cell. In that output, we see the expected module paths on a positive edge of the clock `CK` for the output `Q` and the inverted output `QN`. Furthermore, we see the expected module paths for negative edges of inputs `RN` and `SN` on the two outputs, which are a negated reset and a negated set, respectively. However, what at first seems surprising are the two module paths (**posedge** `RN => (Q +: !SN)`) and (**posedge** `SN => (QN +: !RN)`), which correspond to the deactivation of the reset and set signals, respectively. When looking at the functional description of this cell, one sees that it sets both outputs `Q` and `QN` to `0` if both reset and set are active (i.e., when both `RN` and `SN` are `0`). When either set or reset is deactivated, then the respective other signal is still active, and for just one active signal the outputs `Q` and `QN` are the inverses of each other. Hence, these module paths do exist in the cell, something that could easily be overlooked.

VI. CONCLUSION

We have presented a method for checking module paths against the functional description of cells contained in a cell library. This method is useful to achieve timing closure by identifying infeasible module paths. Furthermore, we developed a technique to extract module paths from a functional description of a cell. This method complements the first one, which allows to remove module paths, by identifying module paths that have not been considered before and therefore would lead to no delay being used. We have implemented both methods and shown that they are applicable to industrial cell libraries.

In the future, we plan to extend the presented methods, especially the derivation of module paths, to other forms of hardware descriptions, such as transistor netlists and behavioral Verilog. Our idea is to extract Boolean equations from a transistor netlist using techniques such as [1] and then performing a similar analysis as that of Section IV. However, since in transistor netlists there are no initial states from which reachability analysis could start, the technique requires some adaptions. For behavioral Verilog we intend to use existing semantics such as [4] to obtain a formal description. However, since these descriptions are often quite large, we plan to use slicing techniques, e.g., [10], to restrict the analysis to only those parts that might contribute to the value of some considered output.

REFERENCES

[1] R. Bryant. Boolean Analysis of MOS Circuits. *IEEE Transactions on Computer-Aided Design*, 6(4):634–649, 1987.

[2] A.J. Daga and W.P. Birmingham. A symbolic-simulation approach to the timing verification of interacting FSMs. In *Proc. of ICCD'95*, pp. 584–589, 1995.

[3] S. Devadas, K. Keutzer, S. Malik, and A. Wang. Certified timing verification and the transition delay of a logic circuit. *IEEE Trans. VLSI Syst*, 2(3):333–342, 1994.

[4] J. Dimitrov. Operational semantics for Verilog. In *Proc. of APSEC'01*, pp. 161–168. 2001.

[5] IEEE Std 1364-2005: IEEE Standard for Verilog Hardware Description Language. IEEE CS, 2006.

[6] Nangate Inc. Open Cell Library v2008_05, 2008. Downloadable from http://www.nangate.com/openlibrary/.

[7] M. Raffelsieper, M.R. Mousavi, J.-W. Roorda, C. Strolenberg, and H. Zantema. Formal Analysis of Non-Determinism in Verilog Cell Library Simulation Models. In *Proc. of FMICS'09*, pp. 133–148, 2009.

[8] M. Raffelsieper, J.-W. Roorda, and M.R. Mousavi. Model Checking Verilog Descriptions of Cell Libraries. In *Proc. of ACSD'09*, pp. 128–137. 2009.

[9] M. Turpin. The Dangers of Living with an X. SNUG Boston, 2003.

[10] S. Vasudevan, E.A. Emerson, and J.A. Abraham. Improved verification of hardware designs through antecedent conditioned slicing. *Softw. Tools for Tech. Transfer*, 9(1):89–101, 2007.