Model Checking Verilog Descriptions of Cell Libraries

Matthias Raffelsieper
Department of Computer Science
TU Eindhoven, P.O. Box 513
Eindhoven, The Netherlands
Email: `M.Raffelsieper@tue.nl`

Jan-Willem Roorda
Fenix Design Automation
P.O. Box 920
Eindhoven, The Netherlands
Email: `janwillem@fenix-da.com`

MohammadReza Mousavi
Department of Computer Science
TU Eindhoven, P.O. Box 513
Eindhoven, The Netherlands
Email: `M.R.Mousavi@tue.nl`

Abstract

We present a formal semantics for a subset of Verilog, commonly used to describe cell libraries, in terms of transition systems. Such transition systems can serve as input to symbolic model checking, for example equivalence checking with a transistor netlist description. We implement our formal semantics as an encoding from the subset of Verilog to the input language of the SMV model-checker. Experiments show that this approach is able to verify complete cell libraries.

1. Introduction

To create digital circuits, designers often use a collection of *standard cells* to implement basic functionalities such as combinational gates (e.g., a NAND or a MUX) and elements that can store a value (e.g., a latch or a flip-flop). Such a collection of standard cells is called a *standard cell library*. Using standard cells allows to split and focus the design effort of larger circuits into functional aspects on the one hand, and facilitates a seamless implementation of the circuit into silicon on the other hand.

In order to enable such a seamless implementation, one wants to have certain guarantees about the functional description of the standard cells, such as for example equivalence with the transistor netlist implementation. These functional descriptions are often given in a subset of the Verilog Hardware Definition Language [1]. Verilog is a standardized language, however there is no standardized semantics for it. Quite a few publications exist that try to fill this gap (e.g., [8], [6], [9]), however these usually address higher level constructs and not the elements that are commonly found in cell libraries, such as built-in and User Defined Primitives (UDPs). An approach covering some aspects of UDPs is reported in [17]. This approach, however, is mainly geared towards an encoding of the Verilog language into gate level networks (via Ordered Ternary Decision Diagrams, OTDDs). To that end, [17] uses heuristics/pattern recognition to detect more complex functions, such as MUXes and XORs in the Verilog description. However, the encoding itself is not formalized and hence, it does not cover many problematic cases, such as multiple inputs to a cell changing at the same time. Furthermore, the goal there was to extract gate level descriptions that are correct-by-construction, whereas our goal is to enable formal verification of Verilog standard cells.

Our research is motivated by a practical problem at Fenix Design Automation. Namely, we want to verify proprietary implementations of cells as transistor netlists against their specifications given in a subset of Verilog consisting of primitives. Traditional equivalence-checking techniques, e.g., based on [7], are not directly applicable to this problem, as they rely on certain structures (e.g., a synchronous gate-level model and a given set of flip-flops) to do matching and to apply retiming. However, in our setting, no such generic structures exist and the elements are custom-made.

In Section 2, we give a formal semantics of the subset of Verilog, which we call VERICELL, that consists of built-in and user defined primitives. Using this semantics, we define an encoding of VERICELL into transition systems in Section 3. A transition system resulting from the encoding can be used as input for symbolic model checkers. In Section 4 we briefly describe how to use these to do equivalence checking. In our experiments, which are presented in Section 5, we show that we could verify complete cell libraries, such as the publicly available Open Cell Library of Nangate [10], as well as proprietary cell libraries provided by Fenix Design Automation. To our knowledge, this is the first result applying formal verification to cell libraries containing UDPs. We conclude the paper in Section 6.

2. Syntax and Semantics of VERICELL

The language VERICELL is a subset of the Verilog Hardware Definition Language, which is defined in the IEEE standard 1364-2005 [1]. This subset consists of the built-in and the user defined primitives (UDPs), and of modules that define the interconnection of these primitives. An example VERICELL program is given in Listing 1, which defines a flip-flop that can be reset.

In the example, we see one module called `flip_flop` that defines the cell. It contains multiple so-called *instantiations* of primitives, for example **`not`**`(ckb, ck)` and `latch(iq, d, ck, rb)`. In such an instantiation, the first

```
module flip_flop (q, d, ck, rb);
  output q; input rb, d, ck;

  not (ckb, ck);
  latch (iq  , d , ck , rb);
  latch (qint, iq, ckb, rb);
  buf (q, qint);
endmodule

primitive latch (Q, D, CK, RB);
output Q; reg Q; input D, CK, RB;
table
//  D      CK      RB      : Qt   :  Qt+1
    0      (?1)    ?       :    ? :    0;
    1      (?1)    1       :    ? :    1;
    ?      (?0)    ?       :    ? :    -;
    ?       *      0       :    0 :    -;
    ?       ?      (?0)    :    ? :    0;
    ?       0      (?1)    :    ? :    -;
    0       1      (?1)    :    0 :    -;
    1       1      (?1)    :    ? :    1;
    *       0      ?       :    ? :    -;
    *       ?      0       :    0 :    -;
   (?0)     1      ?       :    ? :    0;
   (?1)     1      1       :    ? :    1;
endtable
endprimitive
```

Listing 1. Verilog Source of a flip-flop

argument is always the output of the primitive, whereas the remaining arguments are the inputs. The primitives **buf** and **not** are built-in primitives, which copy the input and its negation to the output, respectively. The two instances of `latch` refer to the user defined primitive that is also contained in the program. Such a user defined primitive is defined by giving a truth table to compute the next output value given its input values and the previous value of the output. Other than level values (e.g., `0` or `?`, which match inputs that are 0 or any input, respectively), a table is also allowed to contain transitions of input values, called *edges*. An example of an edge specification is `(?1)`, which describes transitions of an input from an arbitrary value to 1. Another example of an edge specification is `*`, which describes any transition of an input. An edge specification is not allowed for the previous output value. Any row must contain at most one edge specification. A row containing an edge specification is called *edge-sensitive*; otherwise, it is called *level-sensitive*. When some row is applicable to the inputs and to the previous value of the output, then the output value, contained in the last column of that row, is the new output of the UDP. Besides the values `0`, `1`, and `x` this column may also contain the special symbol `-`. This special symbol denotes that the output remains unchanged, i.e., the previous value of the output is also the new output.

The formal syntax of these Verilog elements can be found in [1, Clauses 7, 8, and 12]. Due to space limitations, we will only give a short description of the elements we want to consider.

A VERICELL program contains a single **module** definition which defines the cell. It consists of a name, an interface list, statements designating the output and input signals, and a number of primitive instantiations.

An instantiation of a primitive can only occur inside the only module of the input program. Such an instantiation consists of the name of the primitive followed by a list of identifiers or constants, for example `latch (iq, d, ck, rb)`. Here, the first element in that list is the output of that primitive, in our example this is `iq`. An output identifier may only be the output of one instantiation in the module. After the output, the remaining identifiers or constants are the inputs to the primitive; for our example these are `d`, `ck`, and `rb`. Between the name and the parameters of the instantiated primitive, one is allowed to optionally write a *delay* specification which has the form $\#d$ for some natural number d. Although we have defined our operational semantics for the general case of primitives with delays, we do not consider them here due to space restrictions. Also, in many examples delay specifications are not present, e.g., in the Open Cell Library that we analyzed as a case study these elements do not occur at all. The built-in primitives we allow are **buf**, **not**, **and**, **nand**, **or**, **nor**, **xor**, **xnor**. All of these primitives could also be implemented as UDPs. However, we follow the standard in distinguishing built-in and user defined primitives, since built-in primitives support special syntactic features (e.g., an arbitrary number of inputs) that are not available for UDPs.

We only consider sequential UDPs, i.e., UDPs that may contain edges and may match previous outputs. This is a syntactic restriction, however this does not influence the semantics: any combinational UDP can be converted into a sequential UDP by ignoring the previous value of the output. For sequential UDPs, we accept the full syntax given in the standard, however we will not present the handling of initial output values, which can easily be accommodated.

Given a VERICELL program, we let UDPs (UDPs_n) denote the set of UDPs in the program (that have exactly n inputs). The set Prims denotes the set of all primitives that are used in the program, comprising both UDPs and built-in primitives.

2.1. Semantics of VERICELL

The semantics of VERICELL is defined in an operational style by transforming configurations. In order to define the semantics, we first have to introduce some notations used in the remainder of the section.

All variables in Verilog can have one of the four values Z, 0, 1, or X. However, for the primitives allowed in our subset of Verilog the values Z and X always have the same meaning, representing an unknown value. Therefore, we only consider the *ternary values* $\mathbb{T} = \{0, 1, \mathsf{X}\}$. Here, the values 0 and 1 correspond to the values false and true of the Booleans $\mathbb{B}$, respectively. For the unknown value X, the usual Boolean operations are extended in a pessimistic way, i.e., $\neg\mathsf{X} = \mathsf{X}$, $0 \wedge \mathsf{X} = 0$, $1 \wedge \mathsf{X} = \mathsf{X}$, and $\mathsf{X} \wedge \mathsf{X} = \mathsf{X}$. All other functions on ternary values can be derived from these definitions. A single ternary value $y \in \mathbb{T}$ is also called a *level*, whereas a pair of two ternary values $(y^p, y) \in \mathbb{T} \times \mathbb{T}$ is also called an *edge*.

We first consider the semantics of the primitives and how to compute the output value of these given the values of the inputs and the previous value of the output. This can be expressed by a denotation function. Then we lift this semantics to capture the instantiation of primitives and their interconnections. The latter takes the form of deduction rules.

Output Value of Primitives. For the semantics of built-in primitives we formalize the straight-forward intuitive semantics given in [1, Tables 7-3 and 7-4]. However, no such definition exists for UDPs. Therefore, we first have to define a semantics for these.

The idea of a UDP is to look up the corresponding output value in the table that is given in its declaration. The standard requires that level-sensitive rows take precedence over edge-sensitive rows, i.e., if there are both a level-sensitive and an edge-sensitive row applicable to the current input values, then the output is determined by the level-sensitive row. For example, consider a UDP containing the two rows `(0?) : 0 : 1` and `1 : ? : 0`. If the previous output value of this UDP is 0 and the input changes from 0 to 1 then both rows are applicable. But due to the above requirement the output must always be 0, since this is the output of the level-sensitive row.

However, the standard does not define how to handle the case of multiple inputs changing at the same time. To this end, we compared the outcome of several Verilog simulators such as CVer [5], ModelSim [4], VeriWell [15] and Icarus [16] (unfortunately, some simulators such as Verilator [14] do no support UDPs). Our observation is that the open source simulator CVer and the commercial simulator ModelSim provide the outcome that is consistent with what is specified in the standard and also closest to the intuition of the designers. Hence, we formalize the behavior demonstrated in CVer and ModelSim. (Unless stated otherwise, we formalize the semantics implemented in these two simulators in all other cases of ambiguity in the standard, as well.)

The level specifications `0`, `1`, and `x` that may occur in a truth table of a primitive directly correspond to the ternary values 0, 1, and X, respectively. Therefore, we define for $l \in \{\texttt{0}, \texttt{1}, \texttt{x}\}$ and $y \in \mathbb{T}$ the predicate $\mathrm{match}(l, y)$ to be true if and only if l and y correspond. This is formally defined in Table 1. The additional level specifications `b` and `?` are syntactic sugar, where the first one corresponds to both 0 and 1 and the latter one corresponds to all of the ternary values. Thus, $\mathrm{match}(\texttt{b}, y)$ is true if and only if y is either 0 or 1, whereas $\mathrm{match}(\texttt{?}, y)$ is always true, regardless of the value of y.

An edge specification in a UDP has the general form $\texttt{(}vw\texttt{)}$, where v and w are level specifications. For two values $y^p, y \in \mathbb{T}$ we define for such an edge specification the predicate $\mathrm{match}(\texttt{(}vw\texttt{)}, (y^p, y))$ to be true if and only if $\mathrm{match}(v, y^p)$ and $\mathrm{match}(w, y)$ are true and $y^p \neq y$. This latter requirement is left ambiguous by the standard, however it is enforced by all simulators of Verilog that we tested. The remaining edge specifications `r`, `f`, `p`, `n`, and `*` are expressed using the above and their definitions as given in [1, Table 8-1], as can be seen in Table 1. For example, for the rising specification `r` the predicate $\mathrm{match}(\texttt{r}, (y^p, y))$ is true if and only if $\mathrm{match}(\texttt{(01)}, (y^p, y))$ is true, which in turn is true if and only if y^p is 0 and y is 1, whereas $\mathrm{match}(\texttt{*}, (y^p, y)) = \mathrm{match}(\texttt{(??)}, (y^p, y))$ is true if and only if y^p and y are different values.

We combine the above matching of single values into a predicate matchRow that checks whether a row of a UDP is applicable for a certain vector of inputs. This predicate, whose formal definition is given in Table 1, has as first argument a row of a UDP, in which at most one edge specification may occur. As the second argument, it takes a tuple that contains both levels and edges, where there must be at most one edge. This tuple must have the same length as the number of inputs of the UDP. The last argument of matchRow is the previous output value, which must be a level. It follows from the definition of match that a level specification only matches a level and an edge specification only matches an edge. Therefore, the row matches the inputs if all level inputs have been matched by a level specification and the edge specification, in case the row is edge-sensitive, matches an edge at the same position in the vector of inputs. Furthermore, there must be at most one edge in the inputs, since there is at most one edge specification allowed in a row. To illustrate this, we consider the UDP from Listing 1. For the input values $\texttt{D} = 1$, $\texttt{CK} = (1, 0)$, and $\texttt{RB} = 1$ and the previous output value $\texttt{Q} = 0$ we have for the row in line 16 that $\mathrm{matchRow}(\texttt{? (?0) ? : ? : -}, (1, (1, 0), 1), 0)$ is true. But if we change the input `CK` to be the level 0 and keep the other values, then $\mathrm{matchRow}(\texttt{? (?0) ? : ? : -}, (1, 0, 1), 0)$ is false since $\mathrm{match}(\texttt{(?0)}, 0)$ is false.

The output of a UDP row is given by the ternary value that corresponds to the level specification in the last column. However, also the special symbol "`-`" is allowed there. This value indicates that no change happens, i.e., the output value is the same as the previous output value. Thus, the row `? * 0 : 0 : -` occurring in line 17 of the flip-flop example in Listing 1 could also be written as `? * 0 : 0 : 0`.

Using this and the matching of rows, we want to construct a function that computes the output of a UDP given the

Table 1. Matching of UDP specifications to inputs

$$y^p, y, o^p \in \mathbb{T},\ e \in \mathbb{T}\times\mathbb{T},\ i_1,\dots,i_n \in \mathbb{T}\cup(\mathbb{T}\times\mathbb{T})$$

$$\begin{array}{lcllcl}
\mathrm{match}(\texttt{0}, y) & \doteq & y = \texttt{0} & \mathrm{match}(\texttt{1}, y) & \doteq & y = \texttt{1} \\
\mathrm{match}(\texttt{x}, y) & \doteq & y = \texttt{x} & \mathrm{match}(\texttt{b}, y) & \doteq & \mathrm{match}(\texttt{0}, y) \vee \mathrm{match}(\texttt{1}, y) \\
\mathrm{match}(\texttt{?}, y) & \doteq & \mathsf{true} & & &
\end{array}$$

$$\begin{array}{lcl}
\mathrm{match}(\texttt{(vw)}, (y^p, y)) & \doteq & y^p \neq y \wedge \mathrm{match}(\texttt{v}, y^p) \wedge \mathrm{match}(\texttt{w}, y) \\
\mathrm{match}(\texttt{r}, e) & \doteq & \mathrm{match}(\texttt{(01)}, e) \\
\mathrm{match}(\texttt{f}, e) & \doteq & \mathrm{match}(\texttt{(10)}, e) \\
\mathrm{match}(\texttt{*}, e) & \doteq & \mathrm{match}(\texttt{(??)}, e) \\
\mathrm{match}(\texttt{p}, e) & \doteq & \mathrm{match}(\texttt{(01)}, e) \vee \mathrm{match}(\texttt{(0x)}, e) \vee \mathrm{match}(\texttt{(x1)}, e) \\
\mathrm{match}(\texttt{n}, e) & \doteq & \mathrm{match}(\texttt{(10)}, e) \vee \mathrm{match}(\texttt{(1x)}, e) \vee \mathrm{match}(\texttt{(x0)}, e)
\end{array}$$

$$\mathrm{matchRow}(s_1 \dots s_n : s_{n+1} : o, (i_1,\dots,i_n), o^p) \ \doteq \bigwedge_{1\le j\le n} \mathrm{match}(s_j, i_j) \wedge \mathrm{match}(s_{n+1}, o^p)$$

previous and current values of the inputs and the previous value of the output. However, the standard is not precise enough to allow this computation to be a function. If multiple inputs of a UDP change at the same time, then it is not clear how to handle this. In order for the semantics to be as general as possible, we therefore assume that non-deterministically an order is chosen and according to this order the changed inputs are considered one change at a time. Hence, we get a function that is parametrized by the current UDP, the previous and current values of the inputs, the previous output value, and some order. Such an order is given by a permutation of the numbers from 1 to n, i.e., a bijective function $\pi : \{1,\dots,n\} \to \{1,\dots,n\}$ that dictates the order of checking the inputs whether their values have changed. When we denote all such bijective functions by Π_n, then we can define a function $[\![\cdot]\!] : \mathrm{UDPs}_n \times (\mathbb{T}\times\mathbb{T})^n \times \mathbb{T} \times \Pi_n \times \{1,\dots,n{+}1\} \to \mathbb{T}$ to determine the output value for a UDP $\texttt{udp} \in \mathrm{UDPs}_n$ given some previous and current input values $i_1^p, i_1, \dots, i_n^p, i_n \in \mathbb{T}$, some previous output $o^p \in \mathbb{T}$, and a permutation $\pi \in \Pi_n$ for all $1 \le m \le n$:

$$\begin{array}{lcl}
[\![\texttt{udp}, ((i_1^p, i_1), \dots, (i_n^p, i_n)), o^p, \pi, n+1]\!] & \doteq & o^p \\
[\![\texttt{udp}, ((i_1^p, i_1), \dots, (i_n^p, i_n)), o^p, \pi, m]\!] & \doteq & \\
\multicolumn{3}{l}{[\![\texttt{udp}, ((i_1^p, i_1), \dots, (i_{\pi(m)}, i_{\pi(m)}), \dots, (i_n^p, i_n)), o', \pi, m{+}1]\!]}
\end{array}$$

Here, the next output value o' is defined as follows. If $i^p_{\pi(m)} = i_{\pi(m)}$, then $o' = o^p$, i.e., the value remains unchanged. Otherwise, we define o' to be the corresponding output value of either a level-sensitive row r of the UDP $\texttt{udp}$ for which $\mathrm{matchRow}(r, (i_1^p, \dots, i_{\pi(m)}, \dots, i_n^p), o^p)$ is true or o' is the corresponding output value of an edge-sensitive row r of the UDP for which $\mathrm{matchRow}(r, (i_1^p, \dots, (i^p_{\pi(m)}, i_{\pi(m)}), \dots, i_n^p), o^p)$ is true and for all level-sensitive rows r' of the UDP the property $\mathrm{matchRow}(r', (i_1^p, \dots, i_{\pi(m)}, \dots, i_n^p), o^p)$ is false. If no such row exists, then the next output value o' is defined to be X.

As an example, we consider the primitive `latch` given in Listing 1. We want to determine its output value, when the inputs are `D` $= (1, \mathsf{X})$, `CK` $= (1, 0)$, and `RB` $= (1, 1)$ and the previous output value is `Q` $=$ 0. If we consider the order $\pi = (2, 1, 3)$, then the first intermediate output value is 0, since line 16 satisfies $\mathrm{matchRow}(\texttt{? (?0) ? : ? : -}, (1, (1, 0), 1), 0)$, as we have seen before. Now the previous value of `CK` is updated and the change of input `D` is considered. Here, we see that line 22 matches, giving a next output value `Q` $=$ 0. Finally, since the input `RB` did not change, this is also the output of this instance given these inputs, the previous output, and the considered order.

However, when we change the order to $\pi' = id = (1, 2, 3)$, then we have the following computation which gives the new output value X, showing that the order can change the result of a computation (where `latch` is abbreviated to `l`):

$$\begin{array}{ll}
\ \ [\![\texttt{l}, ((1, \mathsf{X}), (1, 0), (1, 1)), 0, id, 1]\!] & \\
= [\![\texttt{l}, ((\mathsf{X}, \mathsf{X}), (1, 0), (1, 1)), \mathsf{X}, id, 2]\!] & \text{(no match, default)} \\
= [\![\texttt{l}, ((\mathsf{X}, \mathsf{X}), (1, 1), (1, 1)), \mathsf{X}, id, 3]\!] & \text{(line 16 matches)} \\
= [\![\texttt{l}, ((\mathsf{X}, \mathsf{X}), (1, 1), (1, 1)), \mathsf{X}, id, 4]\!] & \text{(}3^{rd}\text{ inp. unchanged)} \\
= \mathsf{X} &
\end{array}$$

Finally, we extend the function $[\![\cdot]\!]$ to incorporate the semantics of built-in primitives. In this way, we get the function $[\![\cdot]\!] : \mathrm{Prims} \times (\mathbb{T}\times\mathbb{T})^n \times \mathbb{T} \times \Pi_n \times \mathbb{N} \to \mathbb{T}$, that uses the informal semantics of the built-in primitives as given in [1, Table 7-3 and Table 7-4]. Note that all built-in primitives are combinational, therefore the order and the previous input values are ignored for them.

Simulation Semantics. Now that we have defined how to evaluate a single primitive, we want to give the semantics of a complete VERICELL program that usually contains a number of instantiations of these primitives. As already noted above, the possible values of variables are considered to be the ternary values $\mathbb{T}$. To keep track of such values in a current configuration, we define a *variable valuation*. A variable valuation val is a partial function mapping identifiers to values from $\mathbb{T}$. If for some identifier x the value $val(x)$ is not defined, then we default to X. We denote a variable valuation as a set of pairs of identifiers and values, e.g., $\{(x, 0)\}$ represents the variable valuation that maps the identifier x to 0. To update variable valuations, we use the operation juxtaposition. Given two variable valuations val_1 and val_2, this operation is defined as $val_1\ \ val_2(x) = val_2(x)$ if $val_2(x)$ is defined, otherwise $val_1\ \ val_2(x) = val_1(x)$. In this way, we have for example $\{(d, 0), (ck, 1)\}\ \ \{(ck, 0), (rb, \mathsf{X})\} = \{(d, 0), (ck, 0), (rb, \mathsf{X})\}$.

The simulation semantics of Verilog is sketched in [1, Clause 11], however it does not deal with the details of when and how to update values. For example, it is not defined when a transition of a variable can be observed by primitives that have this variable as an input. Therefore, when the simulation semantics sketched in [1, Clause 11] is ambiguous, we base our formal semantics on the observations from simulators. As stated before, our choices regarding ambiguities match the interpretation used by CVer and ModelSim.

We split the execution into three different phases, namely, *execute*, *update* and *time-advance*. Execute and update phases are performed iteratively until the current state is stabilized. Only then, the time-advance phase advances the global simulation clock. In the execute phase, all *active* primitives, i.e., primitives for which an input has changed its value, compute their new output values. The output values are stored in a separate location and are not directly used for the evaluation of other primitives, thereby modeling a parallel execution of the primitives. In the update phase, which follows the execute phase, all these values are stored as the new values of the variables. This can again make primitives active, in this case another execute phase is performed.

For example, in the module `flip_flop` shown in Listing 1 a change of the variable `d` might result in a change of the variable `iq`. Since this variable is used as an input to another primitive instantiation with the output variable `qint`, this primitive will be activated and executed. The computation is repeated until no more updates are pending and no primitives are active anymore. Then, the third phase, called time advance phase, is entered in which the global simulation time advances and new inputs are applied. These can again activate primitives in the program, since for example the input `d` might be assigned a different value, so that the execute phase is entered again.

Configurations, i.e., operational states, of our simulation semantics comprise a natural number t, denoting the current

Table 2. Deduction Rules for the Semantics of VERICELL Programs

(ex)
$$\frac{}{\begin{array}{c}\langle t, prev, cur, act \uplus \{\mathtt{p}(o, i_1, \ldots, i_n)\}, up\rangle_E \quad \rightarrow \\ \langle t, prev, cur, act, up\ \ \{(o, [\![\mathtt{p}, ((prev(i_1), cur(i_1)), \\ \ldots, (prev(i_n), cur(i_n))), cur(o), \pi, 1]\!])\}\rangle_E\end{array}}$$

(ex-up)
$$\frac{}{\langle t, prev, cur, \emptyset, up\rangle_E \rightarrow \langle t, cur, cur, \emptyset, up\rangle_U}$$

(up)
$$\frac{up \neq \emptyset}{\begin{array}{c}\langle t, prev, cur, \emptyset, up\rangle_U \rightarrow \\ \langle t, prev, cur\ \ up, sens(cur, cur\ \ up), \emptyset\rangle_U\end{array}}$$

(up-ex)
$$\frac{act \neq \emptyset}{\langle t, prev, cur, act, \emptyset\rangle_U \rightarrow \langle t, prev, cur, act, \emptyset\rangle_E}$$

(time)
$$\frac{}{\begin{array}{c}\langle t, prev, cur, \emptyset, \emptyset\rangle_U \rightarrow \\ \langle t+1, cur, cur\ \ \overrightarrow{in_{t+1}}, sens(\overrightarrow{in_t}, \overrightarrow{in_{t+1}}), \emptyset\rangle_E\end{array}}$$

simulation time, the previous and the current valuations of variables in the module, denoted by $prev$ and cur, respectively. Another component of a configuration is a set act of primitive instantiations that have to be evaluated due to the change of some input. Furthermore, it contains a third variable valuation up that collects the updates to be performed. In order to distinguish the current phase, we introduce a flag called $phase$ that is either E for Execute or U for Update. We do not mention the time-advance phase in $phase$, since it is modelled as a transition relation from an update to an execute phase. Such a configuration is written as $\langle t, prev, cur, act, up\rangle_{phase}$.

Initially, a Verilog program starts in a configuration where the time is 0, all variables have the value X, no primitives are active, and no updates have to be executed, which in our representation is denoted by $\langle 0, \emptyset, \emptyset, \emptyset, \emptyset\rangle_U$. Starting in this initial configuration, the deduction rules presented in the remainder of this section are used to transform a current configuration into a next configuration until this is not possible anymore.

We assume that we are given a sequence $\overrightarrow{in_1}, \ldots, \overrightarrow{in_m}$ of variable valuations called *input vectors*. These variable valuations only assign values to the external inputs declared in the module of the VERICELL program. They are applied whenever the simulation time is allowed to advance.

The operational semantics of a VERICELL program is defined by the deduction rules given in Table 2.

In the execution phase, an active primitive is non-deterministically chosen, executed and removed from the set of active primitives. This is expressed by rule (ex) in Table 2. There, π is an arbitrarily chosen order for the evaluation of inputs. Note that this order may differ for each evaluation of a primitive. The constant 1 denotes that we start with the first value of the permutation π.

When no more active primitives exist then the simulation changes into the update phase. During this change the current transitions of variables are cleared, since all primitives having a changed variable as input have been executed. The new previous values are contained in the *cur* variable valuation, hence this variable valuation will be used as the new previous variable valuation, as can be seen in rule (ex-up) in Table 2.

In the update phase, the new output values are written back into the current variable values. Furthermore, all primitives are activated that have a changed variable as one of their inputs. This is accomplished by the rule (up) in Table 2, where we require $up \neq \emptyset$. The function *sens* computes for two variable valuations the set of primitives that have a changed variable as an input. Formally, this function is defined as $sens(val_1, val_2) = \{\mathtt{p}(o, i_1, \ldots, i_n) \in \text{Prims} \mid \exists 1 \leq j \leq n : val_1(i_j) \neq val_2(i_j)\}$.

When all updates have been performed, then the execute phase is entered again if there are active primitives. This is expressed in the rule (up-ex) in Table 2, which is only applicable if $act \neq \emptyset$. Otherwise, if there are no active primitives, then time advances and the new input values are applied. Furthermore, the new previous values are the current values, because the current state is stable and any present changes can be disregarded. Due to the changed inputs, we determine the primitives that have been activated. For this purpose, the function *sens* is used again in the rule (time) in Table 2.

If no input vector $\overrightarrow{in_{t+1}}$ is available anymore, then the simulation terminates. To determine the trace of output values generated by a certain trace of input vectors, we only consider stable states, i.e., states in which the time can advance or simulation terminates.

To illustrate this semantics, we consider the example given in Listing 1. Let $\overrightarrow{in_1} = \{(\mathtt{d}, 0), (\mathtt{ck}, 1)\}, \overrightarrow{in_2} = \{(\mathtt{d}, 1)\}$ be the external inputs for time steps 1 and 2. Then we get the following simulation, starting in the initial configuration, where we abbreviate `latch` to `l`:

$\langle 0, \emptyset, \emptyset, \emptyset, \emptyset \rangle_U$

Time Advance, applying inputs $\overrightarrow{in_1}$, and activating instantiations

$\rightarrow \langle 1, \emptyset, \{(\mathtt{d}, 0), (\mathtt{ck}, 1)\},$
$\quad \{\mathtt{l(iq,\ d,\ ck,\ rb)}, \mathbf{not}\mathtt{(ckb,\ ck)}\}, \emptyset \rangle_E$

Execution of `l(iq, d, ck, rb)`

$\rightarrow \langle 1, \emptyset, \{(\mathtt{d}, 0), (\mathtt{ck}, 1)\}, \{\mathbf{not}\mathtt{(ckb, ck)}\}, \{(\mathtt{iq}, 0)\} \rangle_E$

Execution of **not**`(ckb,ck)`

$\rightarrow \langle 1, \emptyset, \{(\mathtt{d}, 0), (\mathtt{ck}, 1)\}, \emptyset, \{(\mathtt{iq}, 0), (\mathtt{ckb}, 0)\} \rangle_E$

Execute $\rightarrow$ Update with clearing of edges

$\rightarrow \langle 1, \{(\mathtt{d}, 0), (\mathtt{ck}, 1)\}, \{(\mathtt{d}, 0), (\mathtt{ck}, 1)\},$
$\quad \emptyset, \{(\mathtt{iq}, 0), (\mathtt{ckb}, 0)\} \rangle_U$

Updating values of `iq` and `ckb`, activating `l(qint, iq, ckb, rb)`

$\rightarrow \langle 1, \{(\mathtt{d}, 0), (\mathtt{ck}, 1)\}, \{(\mathtt{d}, 0), (\mathtt{ck}, 1), (\mathtt{iq}, 0), (\mathtt{ckb}, 0)\},$
$\quad \{\mathtt{l(qint,\ iq,\ ckb,\ rb)}\}, \emptyset \rangle_U$

Update $\rightarrow$ Execute

$\rightarrow \langle 1, \{(\mathtt{d}, 0), (\mathtt{ck}, 1)\}, \{(\mathtt{d}, 0), (\mathtt{ck}, 1), (\mathtt{iq}, 0), (\mathtt{ckb}, 0)\},$
$\quad \{\mathtt{l(qint,\ iq,\ ckb,\ rb)}\}, \emptyset \rangle_E$

Execution of `l(qint, iq, ckb, rb)`

$\rightarrow \langle 1, \{(\mathtt{d}, 0), (\mathtt{ck}, 1)\}, \{(\mathtt{d}, 0), (\mathtt{ck}, 1), (\mathtt{iq}, 0), (\mathtt{ckb}, 0)\},$
$\quad \emptyset, \{(\mathtt{qint}, \mathsf{X})\} \rangle_E$

Execute $\rightarrow$ Update, clearing edges

$\rightarrow \langle 1, \{(\mathtt{d}, 0), (\mathtt{ck}, 1), (\mathtt{iq}, 0), (\mathtt{ckb}, 0)\},$
$\quad \{(\mathtt{d}, 0), (\mathtt{ck}, 1), (\mathtt{iq}, 0), (\mathtt{ckb}, 0)\}, \emptyset, \{(\mathtt{qint}, \mathsf{X})\} \rangle_U$

Updating value of `qint` which activates no instantiations

$\rightarrow \langle 1, \{(\mathtt{d}, 0), (\mathtt{ck}, 1), (\mathtt{iq}, 0), (\mathtt{ckb}, 0)\},$
$\quad \{(\mathtt{d}, 0), (\mathtt{ck}, 1), (\mathtt{iq}, 0), (\mathtt{ckb}, 0), (\mathtt{qint}, \mathsf{X})\}, \emptyset, \emptyset \rangle_U$

Time Advance, applying inputs $\overrightarrow{in_2}$, and activating instantiations

$\rightarrow \langle 2, \{(\mathtt{d}, 0), (\mathtt{ck}, 1), (\mathtt{iq}, 0), (\mathtt{ckb}, 0), (\mathtt{qint}, \mathsf{X})\},$
$\quad \{(\mathtt{d}, 1), (\mathtt{ck}, 1), (\mathtt{iq}, 0), (\mathtt{ckb}, 0), (\mathtt{qint}, \mathsf{X})\},$
$\quad \{\mathtt{l(iq,\ d,\ ck,\ rb)}\}, \emptyset \rangle_E$

$\rightarrow \ldots$

$\rightarrow \langle 2, \{(\mathtt{d}, 1), (\mathtt{ck}, 1), (\mathtt{iq}, \mathsf{X}), (\mathtt{ckb}, 0), (\mathtt{qint}, \mathsf{X})\},$
$\quad \{(\mathtt{d}, 1), (\mathtt{ck}, 1), (\mathtt{iq}, \mathsf{X}), (\mathtt{ckb}, 0), (\mathtt{qint}, \mathsf{X})\}, \emptyset, \emptyset \rangle_U$

This simulation terminates in the last configuration, since we did not supply more inputs. The trace of values that we can observe is given by variable valuations representing the current values in the two stable configurations that were reached, these are $\{(\mathtt{d}, 0), (\mathtt{ck}, 1), (\mathtt{iq}, 0), (\mathtt{ckb}, 0), (\mathtt{qint}, \mathsf{X})\}$ and $\{(\mathtt{d}, 1), (\mathtt{ck}, 1), (\mathtt{iq}, \mathsf{X}), (\mathtt{ckb}, 0), (\mathtt{qint}, \mathsf{X})\}$.

3. Encoding VERICELL into Transition Systems

To convert a VERICELL program into a Boolean Transition System (BTS), i.e., a transition system with vectors of Booleans as configuration, we first have to take care of the ternary values. We use a *dual-rail* encoding for these values, where we interpret each variable v as a pair of Boolean variables (v_L, v_H). The first variable v_L represents whether the variable v might be 0, the second variable v_H represents whether v might be 1. Then, we have for the constants in $\mathbb{T}$ that $0 = (\mathsf{true}, \mathsf{false})$, $1 = (\mathsf{false}, \mathsf{true})$, and $\mathsf{X} = (\mathsf{true}, \mathsf{true})$. The fourth value $\mathsf{Z} = (\mathsf{false}, \mathsf{false})$ is considered to be illegal and will never arise as a result of our encodings. For every variable v occurring in the module of the VeriCell program we also introduce another pair of variables $v^p = (v^p_L, v^p_H)$ representing the previous value of the variable.

To combine Boolean and ternary values, we define the implication operation $\rightarrow : \mathbb{B} \times \mathbb{T} \rightarrow \mathbb{T}$ for a ternary value $y \in \mathbb{T}$ as $\mathsf{false} \rightarrow y = \mathsf{X}$ and $\mathsf{true} \rightarrow y = y$. Furthermore, we define a meet operator $\sqcap : \mathbb{T} \times \mathbb{T} \rightarrow \mathbb{T}$ as $(u_L, u_H) \sqcap (v_L, v_H) = (u_L \wedge v_L, u_H \wedge v_H)$ and (in)equality of ternary values as $(u_L, u_H) \neq (v_L, v_H) = \neg\big((u_L, u_H) = (v_L, v_H)\big) = (u_L \oplus v_L) \vee (u_H \oplus v_H)$ for all ternary values $(u_L, u_H), (v_L, v_H) \in \mathbb{T}$. Note that by this definition the value X is a neutral element for $\sqcap$, i.e., $\mathsf{X} \sqcap y = y$ for all ternary values y.

3.1. Encoding of UDPs

To encode the UDPs of a VeriCell program, we first have to encode the matching of row patterns. This can be done in a straightforward way. For level specifications, we define an encoding to match a ternary variable $v = (v_L, v_H)$ as follows:

$$\begin{array}{lcllcl} L(\texttt{0}, v) & \doteq & v_L \wedge \neg v_H & L(\texttt{b}, v) & \doteq & v_L \oplus v_H \\ L(\texttt{1}, v) & \doteq & \neg v_L \wedge v_H & L(\texttt{?}, v) & \doteq & \mathsf{true} \\ L(\texttt{x}, v) & \doteq & v_L \wedge v_H & & & \end{array}$$

For edges, we only have to consider the case of an edge specification of the form `(ab)`, where `a` and `b` are level specifications, since all other edge specifications can be expressed as disjunctions of edge specifications having this shape. The encoding works by simply matching the two level specifications against the previous and the current value of the input. Furthermore, we have to express that really a change has occurred, i.e., the previous and the current value have to be different. This is formalized in the below encoding of an edge specification for pairs v^p, v of ternary variables representing the previous and the current values, respectively:

$$E(\texttt{(ab)}, v^p, v) \quad \doteq \quad (v^p \neq v) \wedge L(\texttt{a}, v^p) \wedge L(\texttt{b}, v)$$

These encodings are now used to encode the evaluation of rows contained in a UDP. If $r = s_1 \ldots s_n : s_{n+1} : s_{n+2}$ is a row from a UDP with n inputs, we let $r|_j$ denote the j-th column of this row, i.e., $r|_j = s_j$ for all $1 \leq j \leq n+2$. For such a row r we define an encoding P that is true if the row matches when considering a changed input value at some position $1 \leq j \leq n$. In case r is a level sensitive row, then we define this encoding as follows, where o is the previous output value, and where $\overrightarrow{i^p} = (i^p_1, \ldots, i^p_n)$ and $\overrightarrow{i} = (i_1, \ldots, i_n)$ are the previous and current inputs, respectively:

$$P(r, o, \overrightarrow{i^p}, \overrightarrow{i}, j) \doteq \bigwedge_{\substack{1 \leq m \leq n \\ m \neq j}} L(r|_m, i^p_m) \wedge L(r|_j, i_j) \wedge L(r|_{n+1}, o)$$

For an edge-sensitive row r, the encoding $P(r, o, \overrightarrow{i^p}, \overrightarrow{i}, j)$ is defined similarly, only there we replace $L(r|_j, i_j)$ by $E(r|_j, i^p_j, i_j)$.

Using this encoding that expresses whether the current row is applicable, we can define the encoding Row that determines an output value w.r.t. some row given the previous and current inputs and the previous output. Here, we have the property that the output of this encoding is X in case the row is not applicable. Again, the number $1 \leq j \leq n$ denotes the position where we consider a change in input values.

$$\mathrm{Row}(r, o, \overrightarrow{i^p}, \overrightarrow{i}, j) \doteq P(r, o, \overrightarrow{i^p}, \overrightarrow{i}, j) \rightarrow O(r, o)$$

Here, the value $O(r, o)$ is the corresponding ternary value to the level specification in the last column of the row, or it is o if the last column of the row contains the symbol "–". As an example, we apply this encoding to the first row of the UDP `latch` in line 14 of Listing 1, where we consider the input `ck` as changed:

$$\begin{array}{l} \mathrm{Row}(\texttt{0 (?1) ? : ? : 0}, \texttt{iq}, (\texttt{d}^p, \texttt{ck}^p, \texttt{rb}^p), \\ \hfill (\texttt{d}, \texttt{ck}, \texttt{rb}), 2) = \\ \quad \texttt{d}^p_L \wedge \neg\texttt{d}^p_H \wedge \big((\texttt{ck}^p_L \oplus \texttt{ck}_L) \vee (\texttt{ck}^p_H \oplus \texttt{ck}_H)\big) \wedge \\ \quad \neg\texttt{ck}_L \wedge \texttt{ck}_H \rightarrow (1, 0) \end{array}$$

In Section 2.1 it was noted that the order of evaluating multiple changed inputs is not fixed by the standard. Hence, there the order was a parameter of the semantics. In our experience, a naive enumeration of all possible orders immediately results in an intractable state-space. In this paper, we fix the order to be the reverse order of inputs as given in the definition of a UDP. This corresponds to our observations of simulators. We are currently working on analysis techniques circumventing this restriction and thus, including all possible orders of execution.

The idea of the encoding is to check recursively for a changed input at the current position of the input. If the currently considered input has changed then the previous output is updated to the new output and the recursion continues for the next position. Otherwise, when the current input is unchanged, then also the output value remains unchanged and the recursion directly advances to the next position.

Formally, we define for a UDP $\mathbf{udp} \in \mathrm{UDPs}_n$ with the output variable o, previous inputs $\overrightarrow{i^p} = (i^p_1, \ldots, i^p_n)$, and current inputs $\overrightarrow{i} = (i_1, \ldots, i_n)$ the encoding $[\![\mathbf{udp}, o, \overrightarrow{i^p}, \overrightarrow{i}]\!]_{\mathbb{B}\times\mathbb{B}} = [\![\mathbf{udp}, o, \overrightarrow{i^p}, \overrightarrow{i}, n]\!]_{\mathbb{B}\times\mathbb{B}}$, which is defined as follows for $1 \leq j \leq n$:

$$\begin{array}{lcl} [\![\mathbf{udp}, o, \overrightarrow{i^p}, \overrightarrow{i}, 0]\!]_{\mathbb{B}\times\mathbb{B}} & \doteq & o \\ [\![\mathbf{udp}, o, \overrightarrow{i^p}, \overrightarrow{i}, j]\!]_{\mathbb{B}\times\mathbb{B}} & \doteq & \\ \multicolumn{3}{r}{\Big((i^p_j = i_j) \rightarrow [\![\mathbf{udp}, o, \overrightarrow{i^p}, \overrightarrow{i}, j-1]\!]_{\mathbb{B}\times\mathbb{B}}\Big)} \\ \multicolumn{3}{r}{\sqcap \Big((i^p_j \neq i_j) \rightarrow [\![\mathbf{udp}, o', \overrightarrow{i^p}', \overrightarrow{i}, j-1]\!]_{\mathbb{B}\times\mathbb{B}}\Big)} \end{array}$$

In the previous values $\overrightarrow{i^p}'$ the changed value i_j replaces the previous value i^p_j, i.e., $\overrightarrow{i^p}' = (i^p_1, \ldots, i_j, \ldots, i^p_n)$. Furthermore, the value of o' is the corresponding value that results from the UDP when considering the change in input j. For this purpose, let $rl_1, \ldots, rl_{k_l}$ be all level-sensitive rows of the UDP and let $re_{1,j}, \ldots, re_{k_e,j}$ be all edge-sensitive rows of the UDP that have an edge specification in column j. Then, the value of o' is defined to be the following:

$$\begin{aligned} o' \;\doteq\; & \prod_{1 \le j_l \le k_l} \mathrm{Row}(rl_{j_l}, o, \overrightarrow{i^p}, \overrightarrow{i}, j) \\ & \sqcap \left[\Big(\neg \bigvee_{1 \le j_l \le k_l} P(p(rl_{j_l}), o, \overrightarrow{i^p}, \overrightarrow{i}, j)\Big) \rightarrow \right. \\ & \left. \qquad \prod_{1 \le j_e \le k_e} \mathrm{Row}(re_{j_e,j}, o, \overrightarrow{i^p}, \overrightarrow{i}, j) \right] \end{aligned}$$

Since this definition consists of meets of implications, the output will be X, i.e., (true, true), if none of the rows is applicable. This is as required in the standard.

3.2. Encoding of Cells

To encode the behavior of a module representing a cell, one could encode the simulation rules given in Section 2.1. However, one can devise a much simpler encoding when restricting to VERICELL programs that do not contain delays. Under this assumption, the new values can directly be written back into the current variables, when we consider the equations that result from the encoding as being evaluated in parallel. This leads to an encoding which only needs the two dual-rail pairs $prev_j$ and cur_j for all of the $n+m$ variables of the module (which is assumed to contain n primitive instantiations and m external inputs) and furthermore m dual-rail pairs inp_j for $n < j \le n+m$ that represent the external inputs to the module.

The next value for a variable $prev_j$ simply has to copy the current value of the variable, so that it represent the previous value in the next iteration. So we define for all $1 \le j \le n+m$ the new value of $prev_j$ as follows:

$$prev_j \;\doteq\; cur_j$$

For the current variables, we have to distinguish whether they represent the output of a primitive instantiation (which we assume to be the first n variables) or an input. For an output of an instantiation p_j, i.e., for $1 \le j \le n$, we define:

$$cur_j \;\doteq\; [\![p_j, cur_j, prev(in(p_j)), cur(in(p_j))]\!]_{\mathbb{B}\times\mathbb{B}}$$

In this definition, we assume that $prev(in(p_j))$ and $cur(in(p_j))$ are vectors of ternary variables and constants that represent the previous and current values of the inputs to p_j, respectively.

We only want to apply new inputs when the current state is stable and time is allowed to advance. Thus, we define a formula $time\_may\_advance$ that is true if the current state is stable. This is the case if there are no more active primitives and no more updates that have to be executed. In our encoding, the updates directly take place in the current variables. Therefore, we only have to detect whether there are no active primitive instantiations. There are no more active instantiations if there are no changed values, i.e., the previous and the current values are the same.

$$time\_may\_advance \;\doteq\; \bigwedge_{1 \le j \le n+m} prev_j = cur_j$$

Using this formula, we can now encode the update of the inputs. The inputs may only be updated if the current state is stable, otherwise the old value has to be kept. This is formalized in the definition below, where $n < j \le n+m$:

$$\begin{aligned} cur_j \;\doteq\; & \neg time\_may\_advance \rightarrow cur_j \\ \sqcap\; & time\_may\_advance \rightarrow [inp_j]_{\mathbb{T}} \end{aligned}$$

In the above definition, $[inp_j]_{\mathbb{T}}$ maps the illegal pair $(\mathsf{false}, \mathsf{false})$ to the pair $(\mathsf{true}, \mathsf{true})$, representing that the value Z behaves like the value X in our subset of Verilog. Formally, we define $[v]_{\mathbb{T}} = (v_L \vee \neg(v_L \wedge v_H), v_H \vee \neg(v_L \wedge v_H))$ for every dual-rail pair $v = (v_L, v_H)$. In this way, $[(\mathsf{false}, \mathsf{false})]_{\mathbb{T}} = (\mathsf{true}, \mathsf{true}) = \mathsf{X}$ and $[y]_{\mathbb{T}} = y$ for every $y \in \mathbb{T}$.

4. Equivalence Checking using LTL Model Checking

The encoding presented in the previous section is used to translate Verilog descriptions into the input language of a symbolic model checker. As an example, we verified equivalence between the Verilog descriptions and the transistor netlist descriptions contained in a cell library. To automatically create a transition system from a transistor netlist, standard techniques exist [2], [12]. These will result directly in a next state function, i.e., when applying such a transition function then the next stable state will be the result of this application.

This is different for the BTSs we create from the Verilog description: Here, we only have a stable state if the time may advance. The property we therefore want to verify is that for all stable states the outputs are equal. Furthermore, we want to restrict the comparison of the outputs to only those where the output is not X, since we regard this as a "dont-care". Finally, we want to make the usual assumption that we have at most one external input changing in every step of the global simulation time. Since not changing the inputs will not trigger any change in a current stable state, we can assume that exactly one input is changing every time the simulation time advances. Note that even with this restriction, there can be multiple inputs of a primitive that can change, due to the parallel execution.

The LTL formula asserting that netlist and Verilog descriptions are equivalent is then expressed as follows, where *outputs* is a set of corresponding pairs of outputs in the Verilog description and the transistor netlist:

$$\big(\mathsf{G}\ \mathit{one\_input\_changes}\big) \rightarrow \Big(\mathsf{G}\ \mathit{time\_may\_advance} \rightarrow \big(\bigwedge_{(o_v,o_t)\in \mathit{outputs}} o_v \neq \mathsf{X} \wedge o_t \neq \mathsf{X} \rightarrow o_v = o_t\big)\Big)$$

The formula expressing that exactly one input changes per timestep, called *one_input_changes* above, is expressed as shown below:

$$\mathit{one\_input\_changes} \doteq \bigvee_{n<j\leq n+m}\Big(([\mathit{inp}_j]_{\mathbb{T}} \neq \mathit{cur}_j) \wedge \bigwedge_{\substack{n<i\leq n+m\\ i\neq j}} ([\mathit{inp}_i]_{\mathbb{T}} = \mathit{cur}_i)\Big)$$

Finally, we also want to add a property stating that always a stable state will eventually be reached. This can be expressed as an LTL formula in the following way, again using the formula *time_may_advance* to indicate whether a current state is stable:

$$\big(\mathsf{G}\ \mathit{one\_input\_changes}\big) \rightarrow \big(\mathsf{G}\ \mathsf{F}\ \mathit{time\_may\_advance}\big)$$

In case we do not want the restriction to input traces where only one input is allowed to change per time step, then we can drop the requirement $(\mathsf{G}\ \mathit{one\_input\_changes})$ from the above formulas.

5. Experiments

As an example set we chose the Nangate Open Cell Library [10] due to its public availability and checked the contained cells for equivalence. We applied an optimized version of the encoding given in Section 3.2 to create BTSs from the Verilog description, and we used an implementation of [2] to extract a transition system from the transistor netlist descriptions. These automatically generated transition systems were then used as input, together with the LTL formulas as described in the previous section, for the symbolic model checkers NuSMV [3] version 2.4.3 and Cadence SMV [11] version 10-11-02p46. NuSMV was run with cone-of-influence reduction and dynamic reordering, whereas we used Cadence SMV in its default settings. All of our experiments were performed on a Pentium 4 with 3 GHz having 1 GB of RAM and running under Linux 2.6.

Almost all cells of the Open Cell Library are described in the VERICELL language, except for the cells `TBUF`, `TINV`, and `TLAT` which use the primitive **`bufif0`**. This primitive is currently not supported, since it distinguishes between X and Z and has non-deterministic behavior for certain input combinations. We also omitted the cells `ANTENNA` and `FILLCELL` which do not implement any functionality. Finally, we did not consider the cells `LOGIC0` and `LOGIC1` which are supposed to provide constant values, however we could not extract a transition system from the transistor netlists given in the cell library.

All considered cells in the Nangate Open Cell Library can be shown equivalent when only changing one input in every time step. Of these 41 cells in the library that we considered, 40 were shown equivalent using Cadence SMV in less than one second. For the last cell, namely the cell called `SDFFRS`, it took about 2.0 seconds to prove equivalence. When using NuSMV, the results are slightly worse, only 36 of the cells were shown equivalent within one second and the proofs of another 4 cells took between 2 and 4 seconds. Again, the cell `SDFFRS` took the most time with 7.8 seconds. If we use optimizations specific to NuSMV for the desired properties, then 38 cells were shown equivalent within one second and 2 of the remaining 3 cells were shown equivalent within 2 seconds. However, the cell `SDFFRS` still required approximately 4.1 seconds.

We also compared the flip-flop example given in Section 2 with the cell `DFFR` from the Nangate Open Cell Library, which also is a flip-flop with reset. However, using our encoding an error trace could be found by both considered model checkers. Such a trace can easily be mapped back to a counterexample in the two cells. This counterexample shows that for the Nangate cell the input already becomes visible at the output at the positive edge of the clock, whereas our flip-flop requires another negative edge. Hence, these two flip-flops are not equivalent and must not be exchanged for one another.

Finally, we ran experiments with a subset of sequential cells taken from a cell-library provided by Fenix Design Automation. For all of these 26 cells we were also able to prove equivalence when considering only one input to change in every time step. Using Cadence SMV, 12 of these cells could be shown equivalent in less than one second. Of the remaining 14 cells, 9 took less than 2 seconds and the remaining 5 cells took between 3 and 15 seconds. For NuSMV the results are comparable when using the optimized encoding of the properties: 12 cells took less than one second, 7 of the remaining 14 cells took less than 2 seconds, and the remaining 7 cells took between 2 and 15.5 seconds.

6. Conclusion

We presented a semantics for the VERICELL subset of Verilog that is commonly used in cell libraries. Using this semantics, we were able to convert VERICELL descriptions automatically into transition systems. These transition systems can be used for model checking, of which we presented an application in the equivalence checking between Verilog descriptions and corresponding transistor netlists contained in a cell library. This check runs fully automatically and requires no expert knowledge. Thereby, one can ensure that the Verilog description exhibits the same behavior as the implementation in silicon.

In the VERICELL subset of Verilog, the values X and Z have the same behavior, as required by the standard. In the future, we would like to extend this subset to also include built-in primitives such as **`bufif0`**. For this purpose we will have to treat the fourth value Z in the logic of Verilog to be different from X for these primitives. However, including these primitives introduces the problem of dealing with non-determinism, since the standard requires for certain input combinations that the output of **`bufif0`** can be either 0 or Z, for example. Also, we want to extend our encoding to be able to deal with delays. As noted previously, delays are already included in our formal semantics for the general case. These rules have to be encoded in the transition systems. Another field we want to investigate further is where the difference in the outputs is only restricted to the "power-up" phase, i.e., in states that will never be reached again. This was already considered by Pixley [13], leading to a different notion of equivalence of transition systems. However, we did not observe these types of counterexamples in the examples we considered. Finally, we have seen that the order of considering changed inputs of UDPs can influence the output of a UDP, thus it can prevent finding counterexamples. This is especially interesting since bugs that stem from such an order dependence may not be found using simulators that use a fixed order. This is part of our current line of research, which includes analyzing non-determinism that can lead to hazards.

Acknowledgments

The authors would like to thank Sandeep Shukla, Chris Strolenberg, Hans Zantema, and Wieger Wesselink for their comments and inspiring discussion.

References

[1] IEEE Std 1364-2005: IEEE Standard for Verilog HDL. IEEE CS, 2006.

[2] R. Bryant. Boolean Analysis of MOS Circuits. *IEEE TCAD*, 6(4):634–649, 1987.

[3] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV Version 2: An OpenSource Tool for Symbolic Model Checking. In *Proc. of CAV'02*, pp. 359–364, 2002.

[4] Mentor Graphics Corp. ModelSim. See http://www.model.com/.

[5] Pragmatic C Software Corp. GPL Cver 2.12a. Downloadable from http://www.pragmatic-c.com/gpl-cver/.

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

[7] C. A. J. van Eijk. Sequential Equivalence Checking Based on Structural Similarities. *IEEE TCAD*, 19(7):814–819, 2000.

[8] M. Gordon. The semantic challenge of Verilog HDL. In *Proc. of LICS'95*, pp. 136–145, 1995.

[9] Z. Huibiao, J. Bowen, and H. Jifeng. From Operational Semantics to Denotational Semantics for Verilog. In *Proc. of CHARME'01*, pp. 449–464, 2001.

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

[11] K. McMillan. A Compositional Rule for Hardware Design Refinement. In *Proc. of CAV'97*, pp. 24–35, 1997.

[12] M. Pandey, A. Jain, R. Bryant, D. Beatty, G. York, and S. Jain. Extraction of finite state machines from transistor netlists by symbolic simulation. In *Proc. of ICCD'95*, pp. 596–601, 1995.

[13] Carl Pixley. A Theory and Implementation of Sequential Hardware Equivalence. *IEEE TCAD*, 11(12):1469–1478, 1992.

[14] W. Snyder. Verilator 3.681, 2008. Downloadable from http://www.veripool.org/wiki/verilator.

[15] WellSpring Solutions. VeriWell 2.8.7. Downloadable from http://sourceforge.net/projects/veriwell.

[16] S. Wilson. Icarus Verilog v0.8.6, 2007. Downloadable from http://www.icarus.com/eda/verilog/.

[17] P. Wohl and J. Waicukauski. Extracting Gate-Level Networks from Simulation Tables. In *Proc. of ITC'98*, pp. 622–631, 1998.