# Formal Analysis of Non-Determinism in Verilog Cell Library Simulation Models

Matthias Raffelsieper[1], MohammadReza Mousavi[1], Jan-Willem Roorda[2], Chris Strolenberg[2], and Hans Zantema[1,3]

[1] Department of Computer Science, TU Eindhoven,
P.O. Box 513, Eindhoven, The Netherlands
Email: {M.Raffelsieper,M.R.Mousavi,H.Zantema}@tue.nl
[2] Fenix Design Automation,
P.O. Box 920, Eindhoven, The Netherlands
Email: {janwillem,chris}@fenix-da.com
[3] Institute for Computing and Information Sciences, Radboud University
P.O. Box 9010, Nijmegen, The Netherlands

**Abstract.** Cell libraries often contain a simulation model in a system design language, such as Verilog. These languages usually involve non-determinism, which in turn, poses a challenge to their validation. Simulators often resolve such problems by using certain rules to make the specification deterministic. This however is not justified by the behavior of the hardware that is to be modeled. Hence, simulation might not be able to detect certain errors. In this paper we develop a technique to prove whether non-determinism does not affect the behavior of the simulation model, or whether there exists a situation in which the simulation model might produce different results. To make our technique efficient, we show that the global property of equal behavior for all possible evaluations is equivalent to checking only a certain local property.

## 1 Introduction

System description languages such as (System)Verilog and SystemC provide several abstraction layers for specifying designs. At higher levels of abstraction, such languages allow for designs with non-deterministic behavior. Although this facility is desirable for high-level designs, it poses a serious challenge for their validation. The common practice in hardware design is to use dynamic validation using simulation kernels, which in turn usually fix a scheduler (i.e., fix several, otherwise arbitrary, parameters) in order to obtain an execution trace of the system. As a result, many plausible runs of the system may be hidden during the validation phase but only show up in the subsequent lower layers and thus, jeopardize the correctness of the final outcome.

An exhaustive search of all possible non-deterministic behavior, using symbolic model-checking techniques, can theoretically solve this challenge. However, in most practical cases, taking all combinations of non-deterministic behavior of components leads to an intractable (symbolic) state space. To alleviate this problem, two main techniques are used: language-based techniques, which make use of language (e.g., Verilog) constructs to rule out irrelevant/impossible combinations at design time and reduction techniques, which propose efficient algorithms to explore only a fraction of the state space while providing sound verification results. The aim of the present paper is to tackle this challenge by combining the above-mentioned techniques in the verification of Verilog cell libraries.

The main motivation for this paper comes from our ongoing cooperation with Fenix Design Automation on the verification of cell libraries. In our earlier publication [8], we report on a formal semantics for the subset of Verilog used in cell libraries. There we observed that the IEEE Standard for Verilog [1], allows for non-deterministic behavior, due to the unspecified order of processing input changes (in case of simultaneous changes in the inputs). Tackling this hugely non-deterministic structure naively (by using a brute-force search) is bound for failure according to our past experience.

In this paper, we propose exhaustive analysis techniques for Verilog cell libraries while addressing their non-deterministic behavior. Our approach is inspired by confluence-checking and confluence reduction techniques from term rewriting and makes use of Verilog timing checks (taking into account constructs such as `$hold` and `$recovery`). Although we develop and apply our techniques to the verification of cell libraries in Verilog, the general problem addressed in this paper is ubiquitous in system design and thus, the techniques can be adapted to and adopted for other domains and input languages. To aid this, we abstract from the exact implementation of evaluating user defined primitives in Verilog and fix only the structure of the computation and two intuitive properties.

**Related work.** In [5], an application of dynamic partial-order reduction techniques is used to efficiently explore all possible execution runs of a test-suite for parallel SystemC processes. To this end, the code of parallel SystemC processes is analyzed and non-commutative transitions are detected. Subsequently, all possible permutations of non-commutative actions are considered in order to generate all schedules that may possibly lead to different final states. The technique reported in [5] is comparable to our confluence-detection and -reduction technique (used in [5] for the purpose of testing instead of exhaustive model-checking). The input language considered in [5] is very rich and hence to cater for the dynamic communication structure of parallel processes some manual code instrumentation is required there, which may be a restrictive factor in industrial cases. In [6], the approach of [5] is enhanced with slicing techniques and combined with static partial order reduction techniques.

Neither of the approaches reported in [5, 6] claim the minimality of the generated schedules. Our approach, however, guarantees that for each two generated schedules, they do produce different output from some initial state and thus, there is a formal justification for including both.

In [4], the technique of [5] is extended to test the vulnerability of a system against changes in the timing specification. To this end, they consider the deviation in timing specification as an ordinary set of inputs for the test process, and thus, check whether a certain choice of deviation for timing specification can result in a new schedule (order of execution), which was not possible be-

fore. Conceptually, this might be considered as dual to our approach, where we make sure that extra, possibly erroneous, execution traces are ruled out by an appropriate timing specification.
**Structure of the paper.** In Section 2, we recall some preliminary concepts. In Section 3, we introduce our basic analysis techniques, namely, commuting diamond analysis and timing checks. In Section 4, we show how to combine these techniques and use model-checking in order to generate concrete counter-examples witnessing all sources of non-determinism in a Verilog cell library. In Section 5, we report on the result of experiments with open source and proprietary cell libraries. Section 6 concludes the paper.

## 2 Preliminaries

### 2.1 Basic Concepts

*Permutations and Lists.* We let $\Pi_n$ denote the set of all *permutations* on the set $\{1, \ldots, n\}$, that is, all bijective functions from $\{1, \ldots, n\}$ to $\{1, \ldots, n\}$. Composition of permutations is denoted by juxtaposition, where $(\pi_1\, \pi_2)\,(x) = \pi_1(\pi_2(x))$. The identity permutation is denoted by id. It is well known that every permutation can be expressed as the composition of *adjacent transpositions*. A transposition is a permutation denoted $(a\,b)$ and defined as $(a\,b)(a) = b$, $(a\,b)(b) = a$, and $(a\,b)(i) = i$ for all $1 \le i \le n$, $i \notin \{a, b\}$. $(a\,b)$ is called *adjacent* if $b = a + 1$.

The set $\overline{\Pi}_n$ denotes the set of all *lists* of numbers from 1 to $n$ which do not contain duplicates. A list $\ell \in \overline{\Pi}_n$ is denoted $\ell = j_1 : \ldots : j_k : \mathsf{nil}$, with the constructor $:$ associating to the right. This list $\ell$ is interpreted as the start of a permutation $\pi_\ell$, i.e., an injective function from $\{1, \ldots, k\}$ to $\{1, \ldots n\}$, for which we have $\pi_\ell(m) = j_m$ for all $1 \le m \le k$. The length of a list is denoted $|\ell|$ and is defined as $|j : \ell'| = 1 + |\ell'|$ and $|\mathsf{nil}| = 0$. Therefore, we have that $\ell$ is a permutation if $|\ell| = n$. We identify every permutation $\pi \in \Pi_n$ with the list $\pi(1) : \ldots : \pi(n) : \mathsf{nil}$. A list $\ell = j_1 : \ldots : j_k : \mathsf{nil} \in \overline{\Pi}_n$ can be constructed by *concatenating* two lists $\ell_1, \ell_2 \in \overline{\Pi}_n$, denoted $\ell_1 {+\!\!+} \ell_2 = \ell$, if $\ell_1 = j_1 : \ldots : j_m : \mathsf{nil}$ and $\ell_2 = j_{m+1} : \ldots : j_k : \mathsf{nil}$ for some $1 \le m \le k$.

*User defined primitives (UDPs).* UDPs are the main building blocks in the Verilog specification of cell libraries. They specify the behavior of basic IP blocks in the form of tables defining an output value corresponding to a set of input values (or changes therein). UDPs can be *combinational* or *sequential*, where in the latter the current value of output is not only dependent on inputs but also on the previous value of outputs. Verilog provides different notation for combinational and sequential circuits, with which we deal in our implementation; however, for the sake of uniformity, we only consider sequential UDPs. (Combinational UDPs are of little relevance for our study of non-determinism anyway.) In this setting, combinational UDPs can be considered as sequential UDPs, in which the row corresponding to the previous value of output is arbitrary (denoted by ?). Henceforth, we drop the word sequential and simply write UDP for sequential UDP.

UDPs work on the *ternary values* $\mathbb{T}$, defined as $\{\mathsf{0}, \mathsf{1}, \mathsf{X}\}$. Here, the values $\mathsf{0}$ and $\mathsf{1}$ correspond to the Boolean values $\mathsf{false}$ and $\mathsf{true}$, respectively. The third value $\mathsf{X}$ can be thought of as representing an unknown value, however this is not enforced for UDPs. A *UDP* with $n$ inputs is a set of rows of the shape $i_1 \ldots i_n : o^p : o$, where $o^p, o \in \mathbb{T}$, there exists at most one $j$, with $1 \leq j \leq n$, such that $i_j \in \mathbb{T} \times \mathbb{T}$, and for all $1 \leq k \leq n$ with $k \neq j$, $i_k \in \mathbb{T}$. The input specification for $i_j$ is called an *edge*, the other specifications are called *levels*. Note that at most one edge specification is allowed in a row; hence, multiple changes in the inputs should be handled one by one and by matching against different rows. Furthermore, for each two rows $r_1 = i_1 \ldots i_n : o^p : o_1$ and $r_2 = i_1 \ldots i_n : o^p : o_2$ in a UDP it holds that $o_1 = o_2$, i.e., two rows specifying the same input and previous output should also produce the same output.

The set of UDPs with $n$ inputs is denoted by $\mathrm{UDPs}_n$. Note that in Verilog, row definitions often contain syntactic sugar that allows to combine multiple row specification in a single row. For example, the symbol ? represents all levels (i.e., all of the three values $\mathsf{0}, \mathsf{1}, \mathsf{X}$), whereas the symbol $*$ represents all edges (i.e., all specifications $(v\,w)$ with $v, w \in \mathbb{T}$ and $v \neq w$). Furthermore, the symbol $-$ can be used in the last column of the row, which indicates the current output value for this row. This symbol stands for no change in the output, i.e., if the value in the previous column, indicating the previous output value, is a value from $\mathbb{T}$, then it can be placed here. A row is called *level-sensitive* if all of its specifications are levels, otherwise, if it contains an edge, it is called *edge-sensitive*. A UDP can be instantiated in a *module* specification.

An example of a sequential UDP, a D Flip-Flop with an enable input, together with a module that instantiates it, is given in Figure 1. A sequential UDP can be distinguished by the keyword **reg**, which declares that the output holds its value between assignments. We use this UDP as our running example throughout the paper. In particular, we check whether prim_ff_en uniquely determines an output function regardless of the order of evaluating its inputs. In other words, we would like to check whether this UDP is *order-independent*.

## 2.2 Semantics of UDPs

We defined the formal semantics of UDPs in [8]. In this section, we briefly recall this semantics and define some notations (for the computation of intermediate states). Furthermore, we state some semantic properties, which are of relevance for the technical developments in the remainder of the paper.

For a UDP *udp* with $n$ inputs, we define the set of all *input vectors* for this UDP to be $I_{udp} = (\mathbb{T} \times \mathbb{T})^n$. We drop the subscript *udp* whenever the considered UDP is clear from the context. Given a UDP and an input vector $\vec{i} = ((i_1^p, i_1), \ldots, (i_n^p, i_n)) \in I$ for it, we define *projections* on it: For $1 \leq k \leq n$, the projections $\rho_k^p, \rho_k : I \to \mathbb{T}$ are defined as $\rho_k^p(\vec{i}) = i_k^p$ and $\rho_k(\vec{i}) = i_k$. The set of all projections for a given UDP *udp* is denoted $\mathrm{Proj}_{udp}$. We drop the subscript when the *udp* is understood from the context. Furthermore, a *substitution* for such a vector is denoted by $\sigma = [a_1^p := v_1, \ldots, a_r^p := v_r, b_1 := w_1, \ldots, b_s := w_s]$ where $a_1, \ldots, a_r, b_1, \ldots, b_s \in \{1, \ldots, n\}$, $v_1, \ldots, v_r, w_1, \ldots, w_s \in \mathbb{T}$, and for every

```
primitive prim_ff_en(q, d, ck, en);
  output q; reg q;
  input d, ck, en;

  table
    //   d   ck    en  :  q  :  q+
         0   (01)  1   :  ?  :  0;
         1   (01)  1   :  ?  :  1;
         ?   (10)  ?   :  ?  :  -;
         *    ?    ?   :  ?  :  -;
         ?    ?    0   :  ?  :  -;
         ?    ?    *   :  ?  :  -;
  endtable
endprimitive


module ff_en(q, d, ck, en);
  output q;
  input d, ck, en;

  prim_ff_en(q, d, ck, en);
endmodule
```

**Fig. 1.** D Flip-Flop with Enable

$1 \leq i, j \leq r$ with $i \neq j$ we have $a_i \neq a_j$ and also for every $1 \leq i, j \leq s$ with $i \neq j$ we have $b_i \neq b_j$. The application of a substitution $\sigma$ to a vector $\vec{i}$ is denoted $\vec{i}\sigma$ and is defined for all $1 \leq k \leq n$ as $\rho^p_k(\vec{i}\sigma) = v$ if $k^p := v \in \sigma$, $\rho_k(\vec{i}\sigma) = w$ if $k := w \in \sigma$, and $\rho^p_k(\vec{i}\sigma) = \rho^p_k(\vec{i})$, $\rho_k(\vec{i}\sigma) = \rho_k(\vec{i})$ in the respective other cases.

We formally defined the semantics of UDP evaluation previously in [8]. This semantics works by considering one input at a time as changed and computing the corresponding next output, which is then used as previous output during the next computation. Intuitively, the next output is computed as follows: First, it is checked whether the considered input has changed. If not, then also the output remains unchanged. Otherwise, the output is determined by looking up a matching row (taking into account that, according to the IEEE standard [1], level-sensitive rows have precedence over edge-sensitive rows) and using its output value. If no such row exists, but the considered input has changed, the next output is set to the default value $\mathsf{X}$.

In this paper, we do not need the full formal definition of the functions, denoted $\Phi_j : I \times \mathbb{T} \to \mathbb{T}$, that are used repeatedly to compute the output when considering the $j$-th input as changed. Instead, we abstract from the concrete implementation in Verilog and only require two properties of these functions.

The first requirement is that when the input considered by such a function is unchanged, then also the output remains unchanged. This clearly holds for Verilog, as can already be seen from the above informal description.

*Property 1.* Let $1 \leq j \leq n$ and let $\vec{i} \in I$ such that $\rho^p_j(\vec{i}) = \rho_j(\vec{i})$. Then for all $o^p \in \mathbb{T}$, $\Phi_j(\vec{i}, o^p) = o^p$.

The second requirement states that the computation of a next output value only depends on the previous values of the inputs, except for the currently considered one. Therefore, one may change a non-considered input of a UDP and will still get the same next output value.

*Property 2.* Let $1 \leq j \leq n$, let $1 \leq k \leq n$ with $k \neq j$, let $\vec{i} \in I$, and let $v, o^p \in \mathbb{T}$. Then $\Phi_j(\vec{i}, o^p) = \Phi_j(\vec{i}[k := v], o^p)$.

Also this property holds for the concrete functions $\Phi_j$ used in Verilog. Intuitively, this is the case since a change on a different position has either already taken place, or it will only take place shortly after the current change.

The semantics of UDP evaluation repeatedly updates the output value, using the above output functions $\Phi_j$ in some specified order. It is defined by the function $[\![\cdot]\!] : \text{UDPs}_n \times I \times \mathbb{T} \times \overline{\Pi}_n \to \mathbb{T}$, which differs slightly from the definition in [8] since in this paper we are also interested in output values after considering only certain inputs, instead of full permutations. Hence, we use a list, instead of a permutation and an index, to identify the input to be considered next. For a $udp \in \text{UDPs}_n$, a vector $\vec{i} \in I$ of previous and current input values, a previous output value $o^p \in \mathbb{T}$, and a list $j : \ell \in \overline{\Pi}_n$ the evaluation is defined recursively:

$$\begin{aligned}
[\![udp, \vec{i}, o^p, \ \mathsf{nil}\ ]\!] &= o^p \\
[\![udp, \vec{i}, o^p, j : \ell]\!] &= [\![udp, \vec{i}[j^p := \rho_j(\vec{i})], \Phi_j(\vec{i}, o^p), \ell\,]\!]
\end{aligned}$$

We will drop the argument *udp* from the above function if the evaluated UDP is clear from the context. Hence, instead of $[\![udp, \vec{i}, o^p, \ell]\!]$ we write $[\![\vec{i}, o^p, \ell]\!]$.

*Order Independence.* A sequential UDP *udp* with $n$ inputs is called *order-independent*, if for all previous and current inputs $\vec{i}$, all previous outputs $o^p$, and all permutations $\pi, \pi' \in \Pi_n$ considered as lists, as defined above, we have $[\![\vec{i}, o^p, \pi]\!] = [\![\vec{i}, o^p, \pi']\!]$. Otherwise, it is called *order-dependent*.

In the example of Figure 1, we have that the order of the inputs d and ck matters for the output of the UDP: For example, if both inputs change from 0 to 1 and the flip-flop is enabled by setting input en to 1, then the value of the output q depends on whether the previous or the current value of input d is used, since in the former case the first row is applicable and sets the output of the UDP to 0, whereas in the latter case the second row is applicable and sets the output to 1. Formally, we have for the UDP prim_ff_en that $[\![((0,1),(0,1),(1,1)), o^p, 2 : 1 : 3 : \mathsf{nil}]\!] = 0 \neq 1 = [\![((0,1),(0,1),(1,1)), o^p, 1 : 2 : 3 : \mathsf{nil}]\!]$ for all previous output values $o^p \in \mathbb{T}$.

## 3 Order Dependency Analysis

### 3.1 Commuting Diamond Analysis

As stated in the preliminaries, a UDP is called order-dependent if two different orders of evaluation lead to different output values. This description leads to a very simple test for order independence, namely to enumerate all pairs of permutations and testing whether the output is the same for all pairs. This naive approach can be slightly improved, by observing that one of the orders can be fixed, for example to the identity permutation.

**Lemma 3.** *A UDP udp* $\in \mathrm{UDPs}_n$ *is order-independent iff* $[\![\vec{i}, o^p, \pi]\!] = [\![\vec{i}, o^p, \mathrm{id}]\!]$ *for all* $\vec{i} \in I$, $o^p \in \mathbb{T}$, $\pi \in \Pi_n$.

*Proof.* The "only if"-direction is trivial from the definition given in the preliminaries. The "if"-part follows from the transitivity of equality. $\square$

This reduces the number of comparisons from $O((n!)^2)$ to $O(n!)$. As we show in the remainder of this section, a quadratic number of comparisons is sufficient to prove order-independence. To this end, we consider pairs of inputs and check whether they satisfy the *commuting diamond property.* Informally, this property expresses that the order of two inputs does not influence the output.

**Definition 4.** *Let udp* $\in \mathrm{UDPs}_n$.
*We say that inputs* $1 \leq i, j \leq n$, $i \neq j$ *have the* commuting diamond property, *denoted* $i \diamond_{udp} j$, *iff for all* $\vec{i} \in I$, $o^p \in \mathbb{T}$:

$$[\![udp, \vec{i}, o^p, i : j : \mathsf{nil}]\!] = [\![udp, \vec{i}, o^p, j : i : \mathsf{nil}]\!]$$

The commuting diamond property is a well-known property from term rewriting, e.g., given in [2, Section 2.7.1]. The idea is that each two one-step rewrites (evaluations) can be joined again by executing the respective other step. Graphically, this is depicted in Figure 2, where only the inputs and the output value are denoted. The solid lines are universally quantified, whereas the dashed lines are existentially quantified.

$$\begin{tikzcd}
 & \vec{i}, o^p \arrow[dl, "i"'] \arrow[dr, "j"] & \\
\vec{i}[i^p := \rho_i(\vec{i})], o_i \arrow[dr, dashed, "j"'] & & \vec{i}[j^p := \rho_j(\vec{i})], o_j \arrow[dl, dashed, "i"] \\
 & \vec{i}[i^p := \rho_i(\vec{i}), j^p := \rho_j(\vec{i})], o &
\end{tikzcd}$$

**Fig. 2.** Commuting Diamond Property

Considering one-step evaluation as a rewrite step, the commuting diamond property implies confluence in the induced term-rewrite system, i.e., the final state (and hence especially the output) is unique regardless of the order of considering inputs. In the sequel we prove a stronger result, namely, that the commuting diamond property and confluence *coincide* in the case of UDP evaluation.

This relies on the semantics of UDPs and does not hold in the general setting of term rewriting. For sake of completeness, we will also include the proof of sufficiency of the commuting diamond property (for the purpose of confluence).

The formal definition of the commuting diamond property amounts to checking that in case of two simultaneous changes in the input, both orders of considering them leads to the same output. To put such evaluations into longer evaluations, where more elements exist in the list of input numbers to be considered, the following lemma shows that this does not change the behavior.

**Lemma 5.** *For a udp* $\in \mathrm{UDPs}_n$*,* $\vec{i} \in I$*,* $o^p \in \mathbb{T}$*, and a list* $\ell_1 \mathbin{+\!\!+} \ell_2 \in \overline{\Pi}_n$ *with* $\ell_1 = k_1 : \ldots : k_{|\ell_1|} : \mathsf{nil}$*:*

$$[\![\vec{i}, o^p, \ell_1 \mathbin{+\!\!+} \ell_2]\!] = [\![\vec{i}[k_r^p := \rho_{k_r}(\vec{i}) \mid 1 \leq r \leq |\ell_1|], [\![\vec{i}, o^p, \ell_1]\!], \ell_2]\!]$$

*Proof.* Induction is performed on $|\ell_1|$.

If $|\ell_1| = 0$, then $\ell_1 = \mathsf{nil}$, $[\![\vec{i}, o^p, \ell_1]\!] = o^p$, and $\vec{i}[k_r^p := \rho_{k_r}(\vec{i}) \mid 1 \leq r \leq |\ell_1|] = \vec{i}$. Hence, $[\![\vec{i}, o^p, \ell_1 \mathbin{+\!\!+} \ell_2]\!] = [\![\vec{i}, o^p, \ell_2]\!] = [\![\vec{i}, [\![\vec{i}, o^p, \ell_1]\!], \ell_2]\!]$.

Otherwise, let $|\ell_1| > 0$ and $\ell_1 = k_1 : \ell$ with $\ell = k_2 : \ldots : k_{|\ell_1|} : \mathsf{nil}$. Then $[\![\vec{i}, o^p, \ell_1]\!] = [\![\vec{i}, o^p, k_1 : \ell]\!] = [\![\vec{i}[k_1^p := \rho_{k_1}(\vec{i})], o', \ell]\!]$ for $o' = \Phi_{k_1}(\vec{i}, o^p)$. Furthermore, $[\![\vec{i}, o^p, \ell_1 \mathbin{+\!\!+} \ell_2]\!] = [\![\vec{i}, o^p, k_1 : \ell \mathbin{+\!\!+} \ell_2]\!] = [\![\vec{i}[k_1^p := \rho_{k_1}(\vec{i})], o', \ell \mathbin{+\!\!+} \ell_2]\!]$. The induction hypothesis is applicable to $\ell$, which proves the theorem:

$$\begin{aligned}
&[\![\vec{i}, o^p, \ell_1 \mathbin{+\!\!+} \ell_2]\!] \\
&= [\![\vec{i}[k_1^p := \rho_{k_1}(\vec{i})], o', \ell \mathbin{+\!\!+} \ell_2]\!] \\
&\overset{\mathrm{IH}}{=} [\![\vec{i}[k_1^p := \rho_{k_1}(\vec{i})][k_r^p := \rho_{k_r}(\vec{i}) \mid 2 \leq r \leq |\ell_1|], [\![\vec{i}[k_1^p := \rho_{k_1}(\vec{i})], o', \ell]\!], \ell_2]\!] \\
&= [\![\vec{i}[k_r^p := \rho_{k_r}(\vec{i}) \mid 1 \leq r \leq |\ell_1|], [\![\vec{i}[k_1^p := \rho_{k_1}(\vec{i})], o', \ell]\!], \ell_2]\!] \\
&= [\![\vec{i}[k_r^p := \rho_{k_r}(\vec{i}) \mid 1 \leq r \leq |\ell_1|], [\![\vec{i}, o^p, \ell_1]\!], \ell_2]\!]
\end{aligned}$$

□

Using the commuting diamond property, we can now show our main lemma. This lemma states that the commuting diamond property is a necessary and sufficient condition to be able to swap the order of two inputs.

**Lemma 6.** *Let udp* $\in \mathrm{UDPs}_n$ *and let* $i : j : \ell \in \overline{\Pi}_n$*.*
*We have* $i \diamond_{udp} j$*, iff for all* $\vec{i} \in I$ *and* $o^p \in \mathbb{T}$*,* $[\![\vec{i}, o^p, i : j : \ell]\!] = [\![\vec{i}, o^p, j : i : \ell]\!]$*.*

*Proof.* In the "only if"-direction, we have the following two computations:

$$[\![\vec{i}, o^p, i : j : \ell]\!] = [\![\vec{i}[i^p := \rho_i(\vec{i}), j^p := \rho_j(\vec{i})], o\ , \ell]\!]$$
$$[\![\vec{i}, o^p, j : i : \ell]\!] = [\![\vec{i}[i^p := \rho_i(\vec{i}), j^p := \rho_j(\vec{i})], o', \ell]\!]$$

Lemma 5 tells us that we can split these computations, and because $i \diamond j$ holds by assumption, we have $o = [\![\vec{i}, o^p, i : j : \mathsf{nil}]\!] = [\![\vec{i}, o^p, j : i : \mathsf{nil}]\!] = o'$. Since the remaining computation is the same, we have shown this direction.

To show the "if"-direction, let $i \not\diamond j$. Then $\vec{i} \in I$ and $o^p, o, o' \in \mathbb{T}$ exist such that:

$$o = [\![\vec{i}, o^p, i : j : \mathsf{nil}]\!] \neq [\![\vec{i}, o^p, j : i : \mathsf{nil}]\!] = o'$$

Define $\vec{i'} = \vec{i}[k := \rho_k^p(\vec{i}) \mid 1 \leq k \leq n, k \notin \{i, j\}]$, i.e., set all current values to their previous values except for those on positions $i$ and $j$. Due to Property 2 we still have $o = [\![\vec{i'}, o^p, i : j : \mathsf{nil}]\!]$ and $o' = [\![\vec{i'}, o^p, j : i : \mathsf{nil}]\!]$. Because of Lemma 5 we have the following two evaluations:

$$\begin{aligned}
[\![\vec{i'}, o^p, i : j : \ell]\!] &= [\![\vec{i'}[i^p := \rho_i(\vec{i}), j^p := \rho_j(\vec{i})], o\ , \ell]\!] \\
[\![\vec{i'}, o^p, j : i : \ell]\!] &= [\![\vec{i'}[i^p := \rho_i(\vec{i}), j^p := \rho_j(\vec{i})], o', \ell]\!]
\end{aligned}$$

By the requirements on lists in $\overline{\Pi}_n$, all remaining elements in $\ell$ are neither $i$ nor $j$. Formally, let $\ell = k_1 : \ldots : k_{|\ell|} : \mathsf{nil}$, then $k_r \notin \{i, j\}$ for all $1 \leq r \leq |\ell|$. Hence, we have $\rho_{k_r}^p(\vec{i'}) = \rho_{k_r}(\vec{i'})$ by construction of $\vec{i'}$ for all $1 \leq r \leq |\ell|$. This allows to repeatedly apply Property 1 to prove this lemma:

$$\begin{aligned}
[\![\vec{i'}, o^p, i : j : \ell]\!] &= [\![\vec{i'}[i^p := \rho_i(\vec{i}), j^p := \rho_j(\vec{i})], o\ , \ell]\!] \\
&= o \\
&\neq o' \\
&= [\![\vec{i'}[i^p := \rho_i(\vec{i}), j^p := \rho_j(\vec{i})], o', \ell]\!] \\
&= [\![\vec{i'}, o^p, j : i : \ell]\!]
\end{aligned}$$

□

Using the above lemma, we can now prove our desired theorem stating that order-independence is equivalent to all pairs of inputs having the commuting diamond property.

**Theorem 7.** *A UDP udp* $\in \mathrm{UDPs}_n$ *with* $n$ *inputs is order-independent, iff for all pairs* $1 \leq i < j \leq n$ *we have* $i \diamond_{udp} j$.

*Proof.* To show the "only if"-direction, let $i \not\diamond j$. Define list $\ell = 1 : \ldots : i - 1 : i + 1 : \ldots : j - 1 : j + 1 : \ldots : n : \mathsf{nil}$. Then by construction both $\pi = i : j : \ell \in \Pi_n$ and $\pi' = j : i : \ell \in \Pi_n$. Lemma 6 tells us that $\vec{i} \in I$ and $o^p \in \mathbb{T}$ exist such that $[\![\vec{i}, o^p, i : j : \ell]\!] \neq [\![\vec{i}, o^p, j : i : \ell]\!]$, which proves that *udp* is order-dependent.

To show the "if"-direction, we assume that $i \diamond j$ for all $1 \leq i < j \leq n$. Let $\pi \in \Pi_n$ with $\pi = (a_1\ a_1{+}1) \cdots (a_k\ a_k{+}1)$. Induction on $k$ is performed to prove the property of Lemma 3.

If $k = 0$, then $\pi = \mathrm{id}$ and hence trivially $[\![\vec{i}, o^p, \pi]\!] = [\![\vec{i}, o^p, \mathrm{id}]\!]$.

Otherwise, let $\pi' = (a_1\ a_1{+}1) \cdots (a_{k-1}\ a_{k-1}{+}1)$. Then for $\vec{i'} = \vec{i}[\pi(r)^p := \rho_{\pi(r)}(\vec{i}) \mid 1 \leq r < a_k]$, $o = [\![\vec{i}, o^p, \pi(1) : \ldots : \pi(a_k{-}1) : \mathsf{nil}]\!]$, and $\ell = \pi(a_k{+}2) : \ldots : \pi(n) : \mathsf{nil}$ we get the following due to Lemmas 5 and 6, since $\pi'(a_k) \diamond \pi'(a_k{+}1)$ by assumption:

$$\begin{aligned}
[\![\vec{i}, o^p, \pi]\!] &= [\![\vec{i}, o^p, \pi'\ (a_k\ a_k{+}1)]\!] \\
&= [\![\vec{i'}, o, \pi'(a_k{+}1) : \pi'(a_k) : \ell]\!] \\
&= [\![\vec{i'}, o, \pi'(a_k) : \pi'(a_k{+}1) : \ell]\!]
\end{aligned}$$

Furthermore, for all $1 \leq m \leq n$ with $m \notin \{a_k, a_k + 1\}$ we have that $\pi(m) = \pi'(m)$. Therefore, by Lemma 5, $[\![\vec{i'}, o, \pi'(a_k) : \pi'(a_k{+}1) : \ell]\!] = [\![\vec{i}, o^p, \pi']\!]$, to which we can apply the induction hypothesis $[\![\vec{i}, o^p, \pi']\!] = [\![\vec{i}, o^p, \mathrm{id}]\!]$ which shows the theorem. □

Coming back to the problem stated at the beginning of this section, we have now a method to check order-independence of UDPs in just $O(n^2)$ function comparisons. To do this, we construct for every pair $1 \leq i < j \leq n$ of inputs the BDDs of the two functions $[\![udp, \vec{i}, o^p, i : j : \mathsf{nil}]\!]$ and $[\![udp, \vec{i}, o^p, j : i : \mathsf{nil}]\!]$, which are then compared for equality. If we have equality of every such pair of functions, then we can conclude order-independence of the UDP, due to the above theorem. If however we find two functions that compute different outputs, then their XOR describes the counterexample states and we have found that the UDP is order-dependent. Furthermore, the construction in the proof of Lemma 6 allows us to conclude that there is a previous output value and an input vector in which only the currently considered inputs are changed that leads to two different outputs depending on the order of the two considered inputs.

When applying this method to the UDP prim_ff_en of Figure 1, then we find, among others, also the example for the input pair d and ck where both inputs change from `0` to `1`, which we already described previously.

For the pair d and en however, no order-dependence exists. This is intuitively true because both changes in d and en will simply keep the current output value, since the output of a Flip-Flop is only changed on a positive edge of the clock.

### 3.2 Verilog Timing Checks

Verilog provides a number of language constructs to specify that critical events (do not) happen within a specified time interval. These constructs are widely used, among others, by the designers of cell libraries for timing specifications that may influence the functional correctness of the designed circuits. The most popular constructs used for this purpose are: `$setup`, `$hold`, `$recovery`, and `$removal`. The syntax of these constructs is given below:

```
$setup/$hold( reference_event, data_event, timing_check_limit
                                            [, notifier] );
```

The `reference_event` is usually an edge of the clock signal (positive, negative or arbitrary, prefixed by `posedge`, `negedge`, or no prefix, respectively). The argument `data_event` is a change in any data signal, and `timing_check_limit` specifies the length of a timing interval in a specified unit of time, e.g., in nanoseconds. Optionally, a `notifier` can be supplied, which is a variable that changes its value whenever the timing check was violated.

The syntax of the `$recovery` and `$removal` timing checks is identical to above, but `reference_event` for these statements denotes an edge of (an asynchronous) control signal. To unclutter the presentation, we only mention `$setup` and `$hold` in the remainder of this paper but the same techniques are applied to `$recovery` and `$removal` constructs.

The semantics of the `$setup` statement enforces that no `data_event` may happen in the (left- and right-) open interval starting `timing_check_limit` before the occurrence of `reference_event` and ending by the occurrence of `reference_event`. The `$hold` statement is dual to `$setup`; it ensures that the `data_event` cannot occur in the left-closed right-open interval starting from the occurrence of `reference_event` and ending `timing_check_limit` time units later. Thus, a pair of `$setup` and `$hold` constructs guarantee a safe margin around any change in the `reference_event` during which the `data_event` cannot occur. In particular, the `$hold` statement prevents the `reference_event` and the `data_event` from happening simultaneously. (Note that the `$setup` statement does not exclude this possibility.)

**Constraints Imposed by Timing Checks.** As stated above, timing checks are added to assert a certain behavior of the system. Otherwise, if this behavior is not encountered, an error is triggered. Hence, for our purposes we can regard the timing checks as describing illegal behavior. Since we are only interested in whether two inputs might change simultaneously, we do not regard the actual time limits nor the notifier variable. We only make use of the restriction that the events of a `$hold` timing check may not occur simultaneously in any execution that is considered legal.

Such constraints can reduce the number of counterexample states for which an order-dependence is found. However, for this to work we have to infer information about the inputs of UDPs from these constraints. The constraints are usually defined on the inputs of the cell which are not necessarily the inputs of the UDP.

If the output of another UDP is used as input to the UDP that is currently checked for order-independence, then we handle this case by making this internal signal a new input of the module. This input might therefore exhibit behavior that is not possible in the implementation, i.e., we might find an order-dependence that does not occur in the implementation.

For the combinational logic driving the inputs of the currently considered UDP, we require that it does not contain loops and we assume that it computes its value instantaneously. Under these assumptions, we can create functions in the external inputs and the outputs of other UDPs (which are now also assumed to be external inputs) and use these as inputs when checking the commuting diamond property. Thereby, we exclude behavior that cannot occur due to functional dependencies of the UDP inputs, and furthermore we get counterexample states that are expressed using these external inputs and the output value of the current UDP.

From these counterexample states we then remove all those states that violate one of the constraints imposed by the `$hold` timing checks. This way, certain input signals of the UDP might become order-independent in all of the allowed executions of the module. Note however, that this order-independence does not solely depend on the UDP anymore, but also especially on the combinational logic and the timing checks present in the module that instantiates the UDP.

To illustrate this, we again consider the UDP prim_ff_en of Figure 1 which admits an order-dependent counterexample for the pair d and ck of inputs, as discussed above. However, this situation is usually considered to be illegal for a D Flip-Flop, hence a designer is likely to add the following timing checks:

```
$hold(posedge ck, negedge d, t1);
$hold(posedge ck, posedge d, t2);
```

These timing checks rule out the behavior leading to the order-dependent counterexample that was described earlier, since the second timing check expresses that it is illegal for the inputs ck and d to change both from `0` to `1` simultaneously. Similarly, all other possible counterexample states involving inputs ck and d are ruled out by these timing constraints, therefore the UDP prim_ff_en has no order-dependency for these two inputs anymore under these constraints.

## 4 Verifying Counterexamples

In the previous sections, we presented how to check order-independence of a UDP and how to restrict this check to only those cases which are not ruled out by the timing specification in the form of `$hold` and `$recovery` timing checks. However, when we report a counterexample this might still be a spurious one. This is due to our overapproximation of UDP outputs and the fact that Verilog has a predetermined initial state, in which all signals have the value X. From this initial state not all counterexample states have to be reachable. Therefore, the idea is to do a reachability analysis, to determine whether a found counterexample is spurious or not.

### 4.1 Required Permutations for Reachability Analysis

Whether a counterexample is spurious or not depends on whether from the initial state one of the counterexample states can be reached or not. However, in contrast to our earlier approach [8], we want to consider all possible execution traces, instead of just those that correspond to the order chosen by the simulator. Our approach is to consider every evaluation of a UDP as independent, i.e., for every evaluation of a UDP the order might be a different one than the order used in another evaluation. This models the behavior of uncontrollable external influences that might determine the order.

Since we want to keep the amount of non-determinism in the generated model as small as possible, we do not generate all orders, but only as many orders as needed for the UDP to exhibit all different behaviors. For this purpose, we use the commuting diamond property presented in Section 3.1 to reduce the number of permutations we have to consider. To this end, we create the set of equivalence classes with respect to the transitive closure of swapping neighboring inputs that have this property. For example, if we have $2 \diamond 3$, then the permutations $2 : 3 : 1 : \mathsf{nil}$ and $3 : 2 : 1 : \mathsf{nil}$ are in the same equivalence class and we only have to consider one of them.

**Definition 8.** *For a UDP udp* $\in \mathrm{UDPs}_n$ *we define a relation* $\leftrightarrow_{udp}$ *on* $\Pi_n$*, where* $\pi \leftrightarrow_{udp} \pi'$ *iff a* $1 \leq k < n$ *exists such that* $\pi = \pi'\,(k\ k{+}1)$ *and* $\pi'(k) \diamond_{udp} \pi'(k+1)$*. Using this relation we define the equivalence relation* $\equiv_{udp}$ *on* $\Pi_n$ *as the reflexive transitive closure of* $\leftrightarrow_{udp}$*.*

This equivalence relation can then be used to partition the set of all permutations into equivalence classes. These equivalence classes still capture all required permutations.

**Lemma 9.** *Let $udp \in \mathrm{UDPs}_n$. For all $\vec{i} \in I$, $o^p \in \mathbb{T}$, and all permutations $\pi \equiv_{udp} \pi' \in \Pi_n$ we have that $[\![\vec{i}, o^p, \pi]\!] = [\![\vec{i}, o^p, \pi']\!]$.*

*Proof.* Let $\pi \equiv \pi'$. Then $\pi = \pi' \, (a_1 \; a_1{+}1) \cdots (a_k \; a_k{+}1)$ for some $a_1, \ldots, a_k \in \{1, \ldots, n-1\}$ with $\pi_{l-1}(a_l) \diamond \pi_{l-1}(a_l + 1)$ for all $1 \le l \le k$, where for every $0 \le l < k$ we define $\pi_l = \pi' \, (a_1 \; a_1{+}1) \cdots (a_l \; a_l{+}1)$. Induction on $k$ is performed.

If $k = 0$, then $\pi = \pi'$, which directly shows the desired property.

Otherwise, $\pi = \pi_{k-1} \, (a_k \; a_k{+}1)$. Because of $\pi_{k-1}(a_k) \diamond \pi_{k-1}(a_k{+}1)$ we can apply Lemmas 5 and 6, which give us for $\vec{i'} = \vec{i}[\pi(r)^p := \rho_{\pi(r)}(\vec{i}) \mid 1 \le r < a_k]$, $o = [\![\vec{i}, o^p, \pi(1) : \ldots : \pi(a_k{-}1) : \mathsf{nil}]\!]$, and $\ell = \pi_{k-1}(a_k{+}2) : \ldots : \pi_{k-1}(n) : \mathsf{nil}$:

$$\begin{aligned}
[\![\vec{i}, o^p, \pi]\!] &= [\![\vec{i}, o^p, \pi_{k-1} \, (a_k \; a_k{+}1)]\!] \\
&= [\![\vec{i'}, o, \pi_{k-1}(a_k{+}1) : \pi_{k-1}(a_k) : \ell]\!] \\
&= [\![\vec{i'}, o, \pi_{k-1}(a_k) : \pi_{k-1}(a_k{+}1) : \ell]\!]
\end{aligned}$$

Furthermore, since $\pi(m) = \pi_{k-1}(m)$ for all $1 \le m < a_k$, we have that $[\![\vec{i'}, o, \pi_{k-1}(a_k) : \pi_{k-1}(a_k{+}1) : \ell]\!] = [\![\vec{i}, o^p, \pi_{k-1}]\!]$ because of Lemma 5. Hence, we can apply the induction hypothesis which gives us $[\![\vec{i}, o^p, \pi_{k-1}]\!] = [\![\vec{i}, o^p, \pi']\!]$. $\square$

Note that above we only use the commuting diamond property and not the `$hold` timing checks. To integrate the latter, we extend the commuting diamond property to a property $\diamond_{udp}^{module}$ by removing counterexample states that were ruled out, as described in the previous section. The resulting equivalence relation is denoted $\equiv_{udp}^{module}$. Also for this relation the above lemma holds, when restricting to only those inputs that do not conflict with the combinational logic and that are not ruled out by a timing check.

These equivalence classes are used in the next section to implement the non-deterministic reachability check. This is done by using only one permutation from each equivalence class, the above lemma tells us that we thereby have considered all possible behaviors of that UDP.

## 4.2 Non-Deterministic Reachability Analysis

In order to check reachability, we follow the approach of [8] and encode the problem as a Boolean Transition System (BTS), which is a transition system with vectors of Booleans as states. However, in contrast to [8], we consider all possible behaviors of the UDPs. For this purpose, we use the required permutations as presented in the previous section and encode the problem in a (non-deterministic) transition relation. This transition relation is defined as the conjunction of the following formulas for each UDP in the cell:

$$\bigvee_{\pi \in \Pi_n / \equiv_{udp}^{module}} \mathrm{next}(o) \leftrightarrow [\![udp, \vec{i}, o, \pi]\!]_{\mathbb{B} \times \mathbb{B}}$$

To make these formulas work on Booleans, we also use a dual-rail encoding of the ternary values, where we define $\mathsf{0} = (\mathsf{true}, \mathsf{false})$, $\mathsf{1} = (\mathsf{false}, \mathsf{true})$, and

$\mathsf{X} = (\mathsf{true}, \mathsf{true})$. Furthermore, $(v_L, v_H) \leftrightarrow (w_L, w_H) = (v_L \leftrightarrow w_L) \wedge (w_L \leftrightarrow w_H)$. The dual-rail encoding $[\![\cdot]\!]_{\mathbb{B}\times\mathbb{B}}$ of UDPs is a straight-forward modification of the dual-rail encoding given in [8], where instead of modeling the order used by simulators we use the order given as extra argument.

Using such a non-deterministic BTS, we can now express the reachability problem in the input language of the SMV model checker. The property we want to verify is the negation of the counterexample states that we want to reach. This way, we get a trace leading to a counterexample state in case an order-dependent UDP can exhibit this behavior in an execution. Note however that we have to restrict the considered traces to the legal traces, as specified by the `$hold` timing checks. This is implemented by adding a state variable *hold_constraints* that is `true` if all states of the currently considered trace have not violated any timing check.

The LTL formula to be checked for a pair $i$ and $j$ of order-dependent inputs is the following, where, by slight abuse of notation, we let $i \not\asymp_{udp}^{module} j$ denote the set of all counterexample states for this pair:

$$\mathsf{G} \neg \left( \mathit{hold\_constraints} \wedge \bigvee_{s \in i \not\asymp_{udp}^{module} j} s \right)$$

As an example, we extend the UDP given in Figure 1 with an asynchronous reset signal as shown in Figure 3. Furthermore, we consider the same `$hold` timing checks for the ck and d inputs that were discussed in Section 3.2.

```
primitive prim_ff_en_rst(q, d, ck, en, rst);
  output q; reg q;
  input d, ck, en, rst;

  table
    //  d   ck   en   rst  :  q  :  q+
        0  (01)  1    ?    :  ?  :  0;
        1  (01)  1    0    :  ?  :  1;
        ?  (10)  ?    0    :  ?  :  -;
        *   ?    ?    0    :  ?  :  -;
        ?   ?    0    0    :  ?  :  -;
        ?   ?    *    0    :  ?  :  -;
        ?   ?    ?    1    :  ?  :  0;
        ?   ?    ?    *    :  0  :  0;
  endtable
endprimitive
```

**Fig. 3.** D Flip-Flop with Enable and Reset

For this UDP our method finds a counterexample for the inputs d and rst. However, this counterexample depends on the previous output value being 1 or X and the input rst having the previous value 1. Such a configuration is not reachable, since setting the input rst to 1 in some previous state always results in the value 0 for the output. This is verified by the SMV model checker, reporting that none of the reachable states is a counterexample state.

No order-dependency between the inputs d and ck is found by our method due to the `$hold` timing checks, as discussed in Section 3.2, and therefore no orders have to be considered which differ in these two inputs.

For the inputs ck and en however, a set of counterexample states is found. When applying the encoding and checking reachability, a trace to a counterexample state is produced, where the previous output is 1, inputs d and rst are 0, and both inputs ck and en change from 0 to 1. This indeed may lead to two different outputs of the UDP, since either the output remains unchanged if the enable signal en is still 0 while the change in the clock ck is processed, or the output takes on the value 0 from the input d if the enable signal is first set to 1 and then the rising edge of ck is considered.

## 5 Experimental Results

To check the applicability of our method, we used it on the Nangate Open Cell Library [7]. It contains 10 different cells that instantiate a sequential UDP and that are in the subset of Verilog studied in this paper.

Using the SMV encoding of the previous section and the NuSMV model checker [3], we found a reachable order-dependent state for all of the cells. However, these counterexamples were due to the value X being allowed as an input of the cell, something that is not possible in a hardware implementation. Hence, we restricted the external inputs to be binary, i.e., to be either 0 or 1. With this restriction, only for 6 cells states exist that can cause an order-dependency. For 4 of these cells none of the counterexample states can be reached, hence the UDPs used in these cells with binary inputs are order-independent.

For the last 2 cells, which are the cells DFFRS and SDFFRS implementing a Flip-Flop (with scan logic) that can be set and reset, a counterexample state can still be reached. The inputs that cause this behavior are in both cases the set and reset inputs. When switching both from active to inactive, the order of this deactivation determines the output of the cell. When deactivating the set signal first, then the reset is still active, forcing the output to be 0. Otherwise, when first deactivating the reset signal, the activated set signal will set the output to be 1. Looking at the Verilog implementation, it seems that for this combination of inputs a `$hold` check was forgotten, since a `$setup` check has been specified. This demonstrates that formal verification of these timing checks is needed and that our method is able to indicate what timing checks might be missing.

We measured the time it took NuSMV to model check reachability of possible counterexample states for both the presented method based on the commuting diamond property and the naive approach based on Lemma 3. It showed that the approach based on the diamond property was consistently faster. Particularly for the largest cell `SDFFRS` the model checking time could be reduced from more than

40 minutes to less than 40 seconds. Also NuSMV's memory consumption was reduced, in the case of the cell `SDFFRS` from more than 880 MB to ca. 110 MB.

Moreover, we have verified a proprietary cell library provided by a client to Fenix Design Automation and found a reachable order dependency there. The reported counterexample is more complex in nature and cannot be traced back to (and possibly even solved by the addition of) missing `$hold` checks. We are investigating other timing specifications / analyses that can generically solve such order dependencies.

## 6 Conclusions

In this paper, we presented formal analysis techniques for detecting nondeterminism in Verilog cell libraries. The source of non-determinism in cell libraries is the arbitrary order of handling multiple changes in inputs. We showed that instead of checking all possible ordering, which is exponential in the number of inputs, it suffices to check the two possible evaluations for each pair of inputs. This approach not only efficiently detects possible sources of non-determinism, but is also complete in that any detected source of non-determinism can lead to two different outputs from some initial state. Our approach is complemented with the language-based control of non-determinism using setup and hold constructs in Verilog. We combined these two approaches and implemented them in a model-checking tool. Open source as well as proprietary cell libraries were analyzed using our implementation and in both cases a number of counterexamples (reachable nondeterministic behavior) were reported using our implementation.

## References

1. IEEE Std 1364-2005: IEEE Standard for Verilog Hardware Description Language. IEEE Computer Society Press, 2006.
2. F. Baader and T. Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998.
3. A. Cimatti et al. NuSMV Version 2: An OpenSource Tool for Symbolic Model Checking. In *Proceedings of CAV'02*, volume 2404 of *Lecture Notes in Computer Science*, pages 359–364, 2002. See also http://nusmv.irst.itc.it.
4. C. Helmstetter, F. Maraninchi, and L. Maillet-Contoz. Test coverage for loose timing annotations. In *Proceedings of FMICS/PDMC'06*, volume 4346 of *Lecture Notes in Computer Science*, pages 100–115. Springer Verlag, 2007.
5. C. Helmstetter, F. Maraninchi, L. Maillet-Contoz, and M. Moy. Automatic generation of schedulings for improving the test coverage of Systems-on-a-Chip. In *Proceedings of FMCAD'06*, pages 171–178. IEEE Computer Society Press, 2006.
6. S. Kundu, M.K. Ganai, and R. Gupta. Partial order reduction for scalable testing of SystemC TLM designs. In *Proc. of DAC'08*, pages 936–941. ACM Press, 2008.
7. Nangate Inc. Open Cell Library v2008_05, 2008. Downloadable from http://www.nangate.com/openlibrary/.
8. M. Raffelsieper, J.-W. Roorda, and M. R. Mousavi. Model Checking Verilog Descriptions of Cell Libraries. In *Proceedings of ACSD'09*, pages 128–137. IEEE Computer Society Press, 2009.