Use of VHDL Environment for Interactive Synthesis of Asynchronous Circuits

N.A.Starodoubtsev A.V.Yakovlev [*] S.Yu.Petrov

Abstract

The paper describes the idea of VHDL-based synthesis of asynchronous control circuits from Signal Transition Graphs. A set of VHDL representation forms is defined which allow the designer: (1) to interact with the synthesis process more closely, using the full range of tools (simulation, visualization etc.) available in VHDL environments; (2) to transition between the forms smoothly, sometimes using "mixed" forms, thus efficiently combining partial logic circuit implementations with event-based descriptions of the surrounding parts.

Topics: Synthesis (VHDL in Synthesis); Specification and modeling methodologies; Simulation

1 Introduction

There has been a rise of interest in asynchronous circuit design in the last decade. As IC features scale down and the speed of components grows, it becomes more problematic to distribute clock across the entire chip without slowing it down in order to mitigate the effect of skew. Furthermore, asynchronous circuits have lent themselves to low power applications [11, 1] due to their inherent ability to operate "on change", that is save power when input data does not change.

There are however certain difficulties with which asynchronous design emerges as a common way to synthesize and implement VLSI logic. A set of recently developed methods and software tools for asynchronous (*control*) logic design can be subdivided into two major groups. One is the so-called circuit compilation approach, the most vivid example being K. van Berkel's VLSI programming using TANGRAM [12]. The latter is essentially a syntax-directed translation of a process-algebraic behavioural specification into an interconnection of building blocks called handshake circuits. This method produces implementations that are delay-insensitive (operate independently of gate and wire delays) or quasi-delay-insensitive (dependent on wire delays though assume that the wire forks are isochronic). The other group forms a logic synthesis approach, in which the implementation is extracted from its behavioural description by means of deriving logic equations, normally through the state graph and logic minimization. This approach is exemplified by synthesis from Signal Transition Graphs (STGs) and related models (related tools include SIS [7], ASSASSIN [13] and FORCAGE [2]). Within this method one can either obtain a speed-independent circuit [2] or a circuit operating correctly under bounded gate and wire delays [7].

[*]Partially supported by EPSRC (GR/J52327) and Royal Society (Joint Project Grant 638072.P688).

The direct compilation approach often produces implementations which are area and time inefficient. The price paid for optimality attainable within the logic synthesis approach is complexity of synthesis procedures. The latter involve solving the so-called Complete State Coding (CSC) problem [4] and finding decomposition of complex gates into the logic architecture available in a particular implementation technology. In practice, the *real* efficiency (as opposed to a potential one) can only be gained if the designer admits to take a more active role in the synthesis process. For example, when solving the CSC problem for an STG model of a circuit, the designer, to overcome certain limitations of the tools, needs to be able to manipulate intermediate representations, e.g., insert new state signals, constrain concurrency, change the order of signal transitions, simulate and visualize their dynamic behaviour under specific delay settings etc. The existing methods and tools suffer from being too specific to their formalisms and internal problems. These tools also have difficulty in working with mixed descriptions, like partial logic implementations surrounded by behavioural (event-based) parts. The latter is important, for example, in designing circuits with logic gates and (internally analogue) mutual exclusion elements.

There exists one more reason of our interest in new descriptive means. Since we permit insertion of new state signals, why cannot we also manipulate their quantity and placement to optimize the implementation? Sometimes even increasing the number of new signals can result in smaller (as in the example below) or faster circuit, as well as in its ability to fit in the given element library. This idea, published in [8, 9], cannot be successfully developed without use of graphical and/or textual descriptions supported by interactive transformations. In this paper, we discuss only the means of textual description.

An average logic designer invariably gets accustomed to the descriptive means supported by the commercial tools of wider usage. Most such tools today extensively rely on VHDL as a high-level behavioural and structural capture language. We therefore perceive that a way to wider acceptance of asynchronous logic synthesis tools lies through a more standardized interface to them. This paper presents one such attempt by embedding a series of representations inherent in the process of synthesizing logic from an STG model into a set of *VHDL forms*. Such forms can be put into an order allowing a flexible transfer from one form to another. They reflect the natural process of converting a *truly event-based view* of the synthesized system, typical for the initial stage of the logic design, to its *functional or structural* view. The forms have been implemented within a package called Vtaxogram, tailored to a VHDL environment through a number of extensions, such as the VHDL function *event* (see Section 3.2). All forms allow the use of standard VHDL facilities, such as compiler, simulator, hierarchical navigator etc. We illustrate these forms and links between them using a fairly simple example (from the SIS benchmark [1]) of a 2-phase-to-4-phase handshake convertor. Most of the STGs examples from the SIS benchmark have been successfully synthesized by inserting some auxiliary signals. Analysis of VHDL specifications for these examples demonstrates that VHDL environment can be successfully used by the designer during interactive synthesis.

The remainder of the paper is organized as follows. Section 2 provides some background for the problems of logic synthesis from STG-like specifications. Section 3 outlines the use of VHDL in logic synthesis, as facilitated by Vtaxogram. Section 4 introduces the VHDL forms and their place in the conversion of STGs to logic implementations. Con-

[1]See, e.g., **converta** in [4].

clusions are drawn in Section 5.

2 Background: Logic Synthesis from STGs

We briefly and informally introduce Signal Transition Graphs (for a more comprehensive introduction see, e.g., [4]). A Signal Transition Graph (STG) is a graphical model of behaviour of an asynchronous control circuit. This model is basically an *interpreted Petri net*, whose transitions (also called events) are annotated with (input/output) signal transitions in the form $+X$ and $-X$, which stand for the *rising* and *falling* edge of a binary signal X. Places of such a Petri net are often shown implicitly in STGs, simply by corresponding flow-relation arcs, if the place have one predecessor and one successor transition. In the latter case, the marking of the place is shown by putting a token directly on its arc. An STG, as a Petri net, generates dynamic behaviour by means of firing transitions and changing its marking. A transition is enabled when each predecessor place (or arc in the "shorthand" notation) has at least one token. An enabled transition may fire, whereby one token is removed from each predecessor place and one token is added on each successor place. An example of STG will be shown in Figure 2(b). STGs are effectively a formal way to depict *timing diagrams*, the latter being a graphical notation for signal waveforms. Timing diagrams are comfortably used by most logic designers but unlike STGs, they are informal and need extra notation or commentary to adequately reflect such paradigms of asynchronous behaviour as concurrency, choice and conflicts. All such paradigms are easily captured by STGs.

It is essential [2] for STG implementation that the state of an STG is completely defined by the marking of its underlying Petri net, whereas the binary encoding of the marking in terms of its signals may be inconsistent or not unique. A *state graph*, produced by an STG, is said to be *consistently encoded* (or simply the STG is consistent) if we can label each state with a binary vector of the STG signals in correspondence with the signs of the labels of transitions enabled in the state. It is known that any consistent STG can be implemented in logic, so that the logic circuit exhibits behaviour equivalent to that of its STG [3]. The implementation may however have additional, internal *state* signals, which must be introduced into the original STG because the latter may not satisfy another part of the implementability condition – *Complete State Coding* (CSC) [4, 3]. An STG is said to violate the CSC condition if it has a pair of different states (markings) with the same encoding but with different non-input signals enabled in them.

The consistency and CSC conditions only guarantee the derivability of Boolean equations from the STG. These equations may however be too complex to be implemented within a particular implementation architecture (logic gate basis). In the latter case more constraints are imposed on the STG and its state graph. To satisfy these constraints, the synthesis procedure must further manipulate the STG, by adding more signals and/or changing the original order of transitions. Modern tools do not fully support ensuring the CSC and other conditions. These problems are generally exponential, but the heuristics used in the tools are often not powerful enough, so the designer must have an easy access to the representations to perform such manipulations. In the following sections we describe a set of VHDL-compatible STG representation forms that have been supported by the Vtaxogram package to facilitate synthesis of asynchronous control circuits in VHDL

[2]Formally, it is the necessary and sufficient condition of STG implementability [4, 3].

environment.

3 Use of VHDL

In this section, we focus our attention on the problem of using VHDL in logic synthesis. As the designer gets closer to the final logic circuit, he or she increasingly begins to think about such pragmatic issues as simulation (e.g., visualizing the functionality of the dynamic behaviour of the models), correction of the derived Boolean functions or circuit, refinement and decomposition (e.g., including, if necessary, some auxiliary transitions and signals). It would be right to use for synthesis the same environment which the designer is accustomed to when working with standard (not necessarily asynchronous) logic design. Thus, we come to the problem of using the VHDL simulator, compiler, hierarchical navigator etc. in asynchronous logic synthesis. The overall framework with which we approach such synthesis is shown in Figure 1. We assume the logic designer eventually to make use of the state-of-the-art STG-based synthesis tools, such as SIS etc., but through the VHDL environment. All theoretically sophisticated algorithms and methods are hidden from the designer by such a "shell" of different representation forms defined in the following sections. The use of the VHDL language is a key point in this approach. A number of interfaces have to be implemented to the actual logic synthesis "engines". At the moment, some of such interfaces are already implemented within the Vtaxogram package.

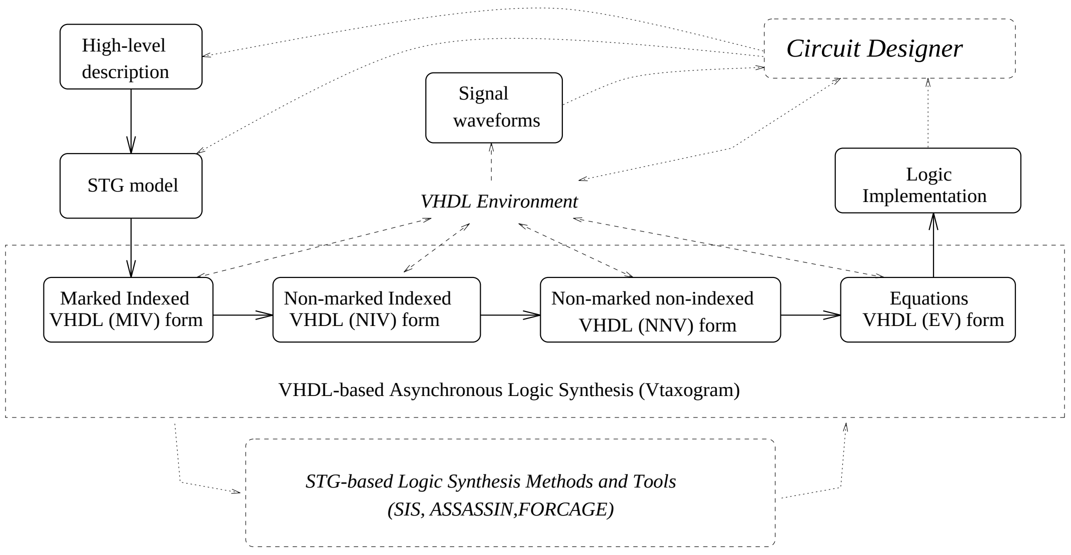
High-level description
Circuit Designer
Signal waveforms
STG model
VHDL Environment
Logic Implementation
Marked Indexed VHDL (MIV) form
Non-marked Indexed VHDL (NIV) form
Non-marked non-indexed VHDL (NNV) form
Equations VHDL (EV) form
VHDL-based Asynchronous Logic Synthesis (Vtaxogram)
STG-based Logic Synthesis Methods and Tools (SIS, ASSASSIN,FORCAGE)

Figure 1: Overview of VHDL-based logic-level synthesis

We assume that the order of signal transitions required to be implemented in the control circuit is specified by an STG, which defines causal relationship between events. For representation of an STG and deriving equations from it we will use, as a basis, VHDL statements of the following format:

$$\text{if} < \mathit{condition} > \text{then} < \mathit{signal\_transitions} > \text{end if;} \tag{1}$$

Note that such representation is somewhat similar to synchronized transitions used in [10] for design.

3.1 Why a single process?

In order to represent an arbitrary concurrent asynchronous behavior all VHDL statements defined by (1) must be executed in parallel, causality thus being determined completely by their conditions. Actually, VHDL gives two alternatives to describe parallelism. We can either use a separate process for each linear sequence of transitions or include all of them into a single process with a sensitivity list. The latter is possible because the order of signal transitions depends on the conditions and delays of transitions rather than on the order of statements (1) inside a process. We prefer to use the second form because the first one may face with some problems of conflict resolution, i.e., when transitions of the same signal take place inside different processes, being *a priori* parallel to each other in VHDL . It does not preclude the designer to use two or more processes (e.g., one for modelling a circuit and the other for an environment).

3.2 Vtaxogram package

We will use assignment statements for the description of signal transition in the field of $< signal\_transitions >$ and use VHDL attribute $'event$ as a part of the guard in the field of $< condition >$ of statement (1). For example, one can use statement

$$\text{if } X'event \text{ and } X = 1 \text{ then } Y <=' 0' \text{ after } d_Y; \text{ end if;} \tag{2}$$

to specify the causal dependence of transition $-Y$ on transition $+X$ where d_Y is the delay of transition $-Y$. Such a description can be shortened with the aid of our VHDL package Vtaxogram, which permits to write

$$\text{if } event(+X) \text{ then } Y <=' 0' \text{ after } d_Y; \text{ end if;} \tag{3}$$

instead of (2) (see also Table 1).

Synchronization and merging points between any two or more control flows are represented in the third and fourth rows of the middle box of Table 1. Note that synchronization corresponds to AND causality, normally defined by STG transitions, while merging to OR causality, defined by STG places. Describing an STG (e.g., see Figure 2) we will also use superscripts (i.e., upper indexes) in order to differentiate one transition $+R_i$ occurrence from another: $+R_i^1$, $+R_i^2$. The Vtaxogram package uses records with two fields to represent *indexed* signal transitions: the first (*value*) is of the bit type , the second (*index*, e.g, the transition's occurrence number) is integer. Typical conditions for indexed signals are represented in the lower box of the table 1.

3.3 "Smooth" transitions between VHDL forms

There are some differences between an STG which specifies the required circuit's behaviour and a set of Boolean equations, specifying the final logic of the synthesized circuit. Firstly, the STG uses marking which acts as a (local) memory to distinguish one state from another. The circuit may be free of any kind of memory, or even without a signal feedback at all. Secondly, it is reasonable to use indexes to differentiate one occurrence of a signal transition from another occurrence. Yet, there is neither a natural equivalent of such an index (which is a truly behavioural identifier rather than functional) in Boolean equations

Table 1: Typical VHDL conditions and their reductions by Vtaxogram package

Original VHDL conditions	The same reduced by Vtaxogram package	Commentaries
$X'event$ and $X =' 0'$	$event(-X)$	Falling edge of X
$X'event$ and $X =' 1'$	$event(+X)$	Rising edge of X
$(X'event$ or $Y'event$ or $\ldots)$ and $(X =' 0'$ and $Y =' 1'$ and $\ldots)$	$event(-X$ and $+Y$ and $\ldots)$	Synchronization $-X$ with $+Y$ and so on
$(X'event$ and $X =' 0')$ or $(Y'event$ and $Y =' 1')$ or $\ldots$	$event(-X$ or $+Y$ or $\ldots)$	Merging $-X$ with $+Y$ and so on
$X'event$ and $X = ('0', i)$	$event(X - i)$	i-th falling edge of X
$X'event$ and $X = ('1', i)$	$event(X + i)$	i-th rising edge of X
$(X'event$ or $Y'event$ or $\ldots)$ and $(X = ('0', i)$ and $Y = ('1', j)$ and $\ldots)$	$event(X - i$ and $Y + j$ and $\ldots)$	Synchronization $-X^i$ with $+Y^j$ and so on
$(X'event$ and $X = ('0', i))$ or $(Y'event$ and $Y = ('1', j))$ or $\ldots$	$event(X - i$ or $Y + j$ or $\ldots)$	Merging $-X^i$ with $+Y^j$ and so on

nor a natural physical mechanism for its implementation, saving perhaps on encoding it by binary signals, which is not always optimal.

The main idea here is to make transformations of an STG to equations smoothly, by gradual getting rid of the "behavioural identification" used in the STG. We thus, firstly, excise the notion of marking and then that of indexing. The designer may interfere into the design process at all transformation steps. Indeed, there is usually a wide choice of possible transformations, with a dramatic impact on the shape of final equations. VHDL is used for all such forms.

Another aspect of interactively assisted conversion is the fact that we ought to provide an opportunity to make such transformations for all signals as mutually independent as possible. For example, the behaviour of one signal could be represented in the marked form with indexes, like in an STG, while some other signals could be represented in the form of Boolean equations.

4 From STG to equations: VHDL forms

In this section we define the main VHDL representation forms showing their role in interactive logic-level synthesis from STGs. For the sake of clarity, we constrain our consideration by a simple example of an STG (see Figure 2). This example does not reflect all possible notational features available in the Vtaxogram package. For example, the full power of STGs also requires use of the VHDL **case** statement, to describe *free (environmental) choice*, and conditions like $X =' 1'$, being sensitive to *levels* of (input) signals (depicted by *self-loop* arcs in STGs) rather than to their transition. We also avoid using an excessive formalism in the sequel.

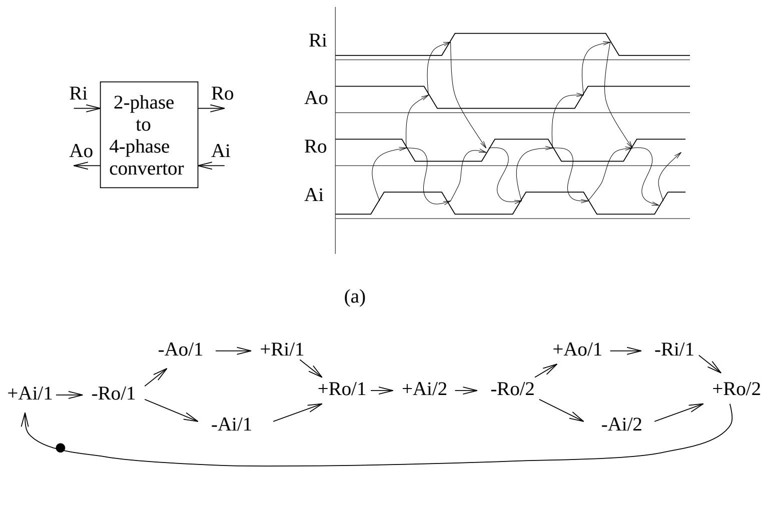

Figure 2: The **converta** example: (a) Structural view and timing diagram; (b) Signal Transition Graph

4.1 Marked indexed VHDL form for STG (MIV)

Under the above constraints, an STG [3] can be considered as a triple of sets: T is the set of transitions like $+X^i$, $-X^i$, where X is name of signal, i is the index (occurrence number) of the signal transition; A is the set of arcs ($A \subseteq T \times T$) being able to be marked; M, ($M \subseteq A$) is the set of initially fired or marked arcs.

The convertor example has two input signals A_i, R_i and two output signals A_o, R_o. Let all transitions of these signals take place inside a single process by the reasons discussed above in Section 3.1. The specification below, which we call the *marked indexed VHDL (MIV) form* or *architecture*, represents the behavior of the STG. (For brevity, we show here only the internals of the VHDL process, and only the first half of it.)

```
if $init'event$       then $arc1$  $<=$ true;                          end if;
if $arc1$             then $A_i$   $<=$ ('1', 1)after $d_{Ai}$;        end if;
if $event(A_i + 1)$   then $arc2$  $<=$ true; $arc1$ $<=$ false;       end if;
if $arc2$             then $R_o$   $<=$ ('0', 1)after $d_{Ro}$;        end if;
if $event(R_o - 1)$   then $arc3$  $<=$ true; $arc6$ $<=$ true;
                           $arc2$  $<=$ false;                         end if;
if $arc6$             then $A_i$   $<=$ ('0', 1)after $d_{Ai}$;        end if;
if $event(A_i - 1)$   then $arc7$  $<=$ true; $arc6$ $<=$ false;       end if;
if $arc3$             then $A_o$   $<=$ ('0', 1)after $d_{Ao}$;        end if;
if $event(A_o - 1)$   then $arc4$  $<=$ true; $arc3$ $<=$ false;       end if;
if $arc4$             then $R_i$   $<=$ ('1', 1)after $d_{Ri}$;        end if;
if $event(R_i + 1)$   then $arc5$  $<=$ true; $arc4$ $<=$ false;       end if;
if $arc5$and$arc7$    then $R_o$   $<=$ ('1', 1)after $d_{Ro}$;        end if;
if $event(R_o + 1)$   then $arc8$  $<=$ true; $arc5$ $<=$ false;
                           $arc7$  $<=$ false;                         end if;
```

[3]This is effectively a subclass of STGs based on Marked Graph Petri nets [4].

In order to initialize the process, signal *init* (from another process) fires $arc1$ in the first statement. Such a specification can be simulated in the VHDL environment under various delays d_{Ai}, d_{Ri}, d_{Ao}, d_{Ro} associated with corresponding signals (later, with logic gates).

This specification may seem a bit bulky and hardly readable, even for such a simple example. We present this example only to show that it is necessary before using other VHDL forms, which are not syntactically close to the initial STG. The single advantage of MIV form is its exact correspondence to STG.

4.2 Non-marked indexed form (NIV)

The explicit notion of marking can be erased by removing (intermediate) variables $arc1, arc2, \ldots$ as well as all corresponding transitions while preserving the initial causality relations. The obtained *architecture* is called the *non-marked indexed VHDL (NIV) form*, with the whole **converta** process looking as follows:

```
if  $event(+init\text{or}R_o+2)$      then  $A_i <= ('1',1)$  after  $d_{Ai}$;  end if;
if  $event(A_i+1)$                     then  $R_o <= ('0',1)$  after  $d_{Ro}$;  end if;
if  $event(R_o-1)$                     then  $A_i <= ('0',1)$  after  $d_{Ai}$;  end if;
if  $event(R_o-1)$                     then  $A_o <= ('0',1)$  after  $d_{Ao}$;  end if;
if  $event(A_o-1)$                     then  $R_i <= ('1',1)$  after  $d_{Ri}$;  end if;
if  $event(R_i+1\text{and}A_i-1)$      then  $R_o <= ('1',1)$  after  $d_{Ro}$;  end if;
if  $event(R_o+1)$                     then  $A_i <= ('1',2)$  after  $d_{Ai}$;  end if;
if  $event(A_i+2)$                     then  $R_o <= ('0',2)$  after  $d_{Ro}$;  end if;
if  $event(R_o-2)$                     then  $A_i <= ('0',2)$  after  $d_{Ai}$;  end if;
if  $event(R_o-2)$                     then  $A_o <= ('1',1)$  after  $d_{Ao}$;  end if;
if  $event(A_o+1)$                     then  $R_i <= ('0',1)$  after  $d_{Ri}$;  end if;
if  $event(R_i-1\text{and}A_i-2)$      then  $R_o <= ('1',2)$  after  $d_{Ro}$;  end if;
```

It is easier for understanding than the MIV form, its first six lines describing the same behaviour as the latter above, which can be verified by simulation under various distributions of delays d_{Ai}, d_{Ri}, d_{Ao}, d_{Ro} using standard VHDL tools. Alternatively, one may examine the causal relations between signal transition events.

Note that similar transformation from the MIV to NIV form may not always be done successfully by simple deletion of arcs in the MIV form. For more complex examples, one may need to add auxiliary signals and their transitions, just as for converting the NIV form to the form considered in the following subsection.

4.3 Non-marked non-indexed form (NNV)

In order to get closer to the final equations we need to get rid of indexes. If we simply try to delete all indexes from the NIV specification, the corresponding architecture will not perform correctly. Indeed, indices much like arcs in the subsection 4.1, bear some form of memory (though a "global" one as opposed to the "local" memory of arcs) in the model's behaviour, helping it to distinguish between semantically different states. It is easy to observe that such an (incorrectly obtained) architecture and the NIV one describe two different behaviours, even under equal delays $d_{Ai} = d_{Ri} = d_{Ao} = d_{Ro}$.

A correct *non-marked non-indexed VHDL (NNV) architecture* is given below, an auxiliary signal X having been inserted in it [4].

```
if  $event(+init\text{or} + R_o)$              then  $A_i <= '1'$  after  $d_{Ai}$;  end if;
if  $A_o = '1'\text{and}event(+A_i)$            then  $X <= '0'$    after  $d_X$;     end if;
if  $event(-X)$                                 then  $R_o <= '0'$  after  $d_{Ro}$;  end if;
if  $X = '0'\text{and}event(-R_o)$              then  $A_i <= '0'$  after  $d_{Ai}$;  end if;
if  $event(-R_o)$                               then  $A_o <= '0'$  after  $d_{Ao}$;  end if;
if  $event(-A_o)$                               then  $R_i <= '1'$  after  $d_{Ri}$;  end if;
if  $X = '0'\text{and}$
    $event(+R_i\text{and} - A_i)$               then  $R_o <= '1'$  after  $d_{Ro}$;  end if;
if  $event(+R_o)$                               then  $A_i <= '1'$  after  $d_{Ai}$;  end if;
if  $A_o = '0'\text{and}event(+A_i)$            then  $X <= '1'$    after  $d_X$;     end if;
if  $event(+X)$                                 then  $R_o <= '0'$  after  $d_{Ro}$;  end if;
if  $event(-R_o)$                               then  $A_i <= '0'$  after  $d_{Ai}$;  end if;
if  $X = '1'\text{and}event(-R_o)$              then  $A_o <= '1'$  after  $d_{Ao}$;  end if;
if  $event(+A_o)$                               then  $R_i <= '0'$  after  $d_{Ri}$;  end if;
if  $X = '1'\text{and}$
    $event(-R_i\text{and} - A_i)$               then  $R_o <= '1'$  after  $d_{Ro}$;  end if;
```

4.4 Equations VHDL form (EV)

NNV is the form where any binary signal transition and hence its value depends only on the values of other input and output signals (and possibly, some additional state signals) within the sensitivity lists of the process. Thus, NNV can be considered as a specific form of incompletely determined Boolean functions. It is ready to generate truth tables for non-input signals (implied values) and then equations in a traditional way [4]. The resulting architecture will be as follows:

```
if  $event(+init\text{or} + R_o)$              then  $A_i <= '1'$  after  $d_{Ai}$;  end if;
if  $X = '0'\text{and}event(-R_o)$              then  $A_i <= '0'$  after  $d_{Ai}$;  end if;
if  $event(-A_o)$                               then  $R_i <= '1'$  after  $d_{Ri}$;  end if;
if  $event(+R_o)$                               then  $A_i <= '1'$  after  $d_{Ai}$;  end if;
if  $event(-R_o)$                               then  $A_i <= '0'$  after  $d_{Ai}$;  end if;
if  $event(+A_o)$                               then  $R_i <= '0'$  after  $d_{Ri}$;  end if;
```

$$
\begin{aligned}
A_o &<= (A_o\text{and}R_o)\text{or}(A_o\text{and}X)\text{or}(\text{not}R_o\text{and}X)\text{after } d_{Ao};\\
R_o &<= (\text{not}A_i\text{and}R_i\text{andnot}X)\text{or}(\text{not}A_i\text{andnot}R_i\text{and}X)\text{or}\\
&\quad (A_o\text{and}R_o\text{and}X)\text{or}(\text{not}A_o\text{and}R_o\text{andnot}X)\text{after } d_{Ro};\\
X &<= (A_i\text{andnot}A_o\text{and}R_o)\text{or}(\text{not}A_i\text{and}X)\text{or}\\
&\quad (R_i\text{and}X)\text{or}(\text{not}R_o\text{and}X)\text{after } d_X;
\end{aligned}
$$

Such a form, where the behavior of all non-input signals is given by their Boolean equations, is called the *equation VHDL (EV) form.* Note that one needn't perform the overall conversion of any initial form to the NNV one to obtain only a partial logic implementation solution. In order to get the equation, say, for signal A_o, it is sufficient to end up at any mixed form, making sure that signal A_o depends only on binary signals. In this case, the following VHDL architecture can be used (where the whole environment of Ao element is described in eventual manner):

[4]We leave out actual theory behind such an insertion; this theory is related to solving the Complete State Coding, discussed, e.g., in [4, 9].

```
if  $event(+init\text{or} + R_o)$          then  $A_i <=' 1'$  after  $d_{Ai}$;  end if;
if  $A_o =' 1'\text{and}event(+A_i)$       then  $X <=' 0'$    after  $d_X$;     end if;
if  $event(-X)$                            then  $R_o <=' 0'$  after  $d_{Ro}$;  end if;
if  $X =' 0'\text{and}event(-R_o)$         then  $A_i <=' 0'$  after  $d_{Ai}$;  end if;
if  $event(-A_o)$                          then  $R_i <=' 1'$  after  $d_{Ri}$;  end if;
if  $X =' 0'\text{and}$
    $event(+R_i\text{and} - A_i)$          then  $R_o <=' 1'$  after  $d_{Ro}$;  end if;
if  $event(+R_o)$                          then  $A_i <=' 1'$  after  $d_{Ai}$;  end if;
if  $A_o =' 0'\text{and}event(+A_i)$       then  $X <=' 1'$    after  $d_X$;     end if;
if  $event(+X)$                            then  $R_o <=' 0'$  after  $d_{Ro}$;  end if;
if  $event(-R_o)$                          then  $A_i <=' 0'$  after  $d_{Ai}$;  end if;
if  $event(+A_o)$                          then  $R_i <=' 0'$  after  $d_{Ri}$;  end if;
if  $X =' 1'\text{and}$
    $event(-R_i\text{and} - A_i)$          then  $R_o <=' 1'$  after  $d_{Ro}$;  end if;
```

$$A_o \quad <= \quad (A_o\text{and}R_o)\text{or}(A_o\text{and}X)\text{or}(\text{not}R_o\text{and}X)\text{after } d_{Ao};$$

4.5 Alternative signals insertions

Here we only illustrate an idea of manipulation with more auxiliary signals. Let us use two such signals X and Y instead of just one signal X, as was done in subsection 4.3. One of their possible placements gives the following new NNV form:

```
if  $event(+init\text{or} + R_o)$          then  $A_i <=' 1'$  after  $d_{Ai}$;  end if;
if  $A_o =' 1'\text{and}event(+A_i)$       then  $X <=' 1'$    after  $d_X$;     end if;
if  $event(+X)$                            then  $R_o <=' 0'$  after  $d_{Ro}$;  end if;
if  $X =' 1'\text{and}event(-R_o)$         then  $A_i <=' 0'$  after  $d_{Ai}$;  end if;
if  $event(-R_o)$                          then  $A_o <=' 0'$  after  $d_{Ao}$;  end if;
if  $event(-A_o)$                          then  $R_i <=' 1'$  after  $d_{Ri}$;  end if;
if  $X =' 1'\text{and}$
    $event(+R_i\text{and} - A_i)$          then  $X <=' 0'$    after  $d_X$;     end if;
if  $event(-X)$                            then  $R_o <=' 1'$  after  $d_{Ro}$;  end if;
if  $event(+R_o)$                          then  $A_i <=' 1'$  after  $d_{Ai}$;  end if;
if  $A_o =' 0'\text{and}event(+A_i)$       then  $Y <=' 1'$    after  $d_Y$;     end if;
if  $event(+Y)$                            then  $R_o <=' 0'$  after  $d_{Ro}$;  end if;
if  $event(-R_o)$                          then  $A_i <=' 0'$  after  $d_{Ai}$;  end if;
if  $Y =' 1'\text{and}event(-R_o)$         then  $A_o <=' 1'$  after  $d_{Ao}$;  end if;
if  $event(+A_o)$                          then  $R_i <=' 0'$  after  $d_{Ri}$;  end if;
if  $Y =' 1'\text{and}$
    $event(-R_i\text{and} - A_i)$          then  $Y <=' 0'$    after  $d_Y$;     end if;
if  $event(-Y)$                            then  $R_o <=' 1'$  after  $d_{Ro}$;  end if;
```

The latter may be transformed to an EV form which is considerably simpler even than the minimal solution obtainable for the case of using a single auxiliary signal X. Only the equation part is shown below, because its event-based part is the same as the one from subsection 4.4 (upper part of the first example).

$$\begin{array}{lcl}
A_o & <= & (\text{not}R_o\text{and}Y)\text{or}(\text{not}X\text{and}A_o)\text{or}(R_o\text{and}A_o)\text{after } d_{Ao}; \\
R_o & <= & \text{not}X\text{andnot}Y\text{after } d_{Ro}; \\
X & <= & (\text{not}R_i\text{and}A_i\text{and}R_o)\text{or}(\text{not}R_i\text{and}X)\text{or}(A_i\text{and}X)\text{after } d_X; \\
Y & <= & (R_i\text{and}A_i\text{and}R_o)\text{or}(R_i\text{and}Y)\text{or}(A_i\text{and}Y)\text{after } d_Y;
\end{array}$$

It illustrates the fact that the "quality" of the obtained solution depends more on how fortunate the "state coding" or, equally, the placement of auxiliary signals is, than on the minimization of Boolean functions. However, this problem is more difficult to formalize than the latter one, which explains our adherence to interactive synthesis, particularly in VHDL environment.

5 Conclusion

We have described a set of equivalent representation forms, making up a notational framework for a VHDL-based logic-level synthesis environment. These forms allow the designer to interfere into the process of logic synthesis from STGs into Boolean functions for non-input signals. The designer may easily represent the circuit's functionality at different levels of "behavioural identification", making use of or excluding the explicit notion of marking (arcs), occurrence numbers (indices) and event-based causality. Using mixed forms, such as NNV and EV, the designer may combine event-based and level-based parts of the design description. Such combinations may be useful for simulating the compositions of logic parts of the circuit with some analogue parts (such as *mutual exclusion elements and arbiters*), whose external behaviour can only be captured at the event-based level. Furthermore, compositions of partial implementations (in EV form) with their surrounding environment (in other forms) can be efficiently modelled and subject to simulation and/or formal verification (e.g., checking for speed-independence [5] or correctness under timing constraints [6]).

The NIV architecture seems to be most convenient for VHDL behavioral descriptions in an event-based (i.e., causal) manner, while the MIV and NNV architectures are useful rather as bridges between STG and equations, respectively. Conversions between one architecture and another can be considered as a sort of specific approach to solving the CSC problem within VHDL environment, the latter being useful for informal estimations of possible insertions of new signals.

A similar approach [8] was used for synthesis of circuits in a specific (antitonic) logic basis, however, without use of VHDL. Note that this approach also avoided working with state-based representations (for the reasons of complexity reductions). Recent applications of the approach to SIS benchmarks indicates an essential source for reducing complexity under equal other conditions. This approach, combined with the VHDL environment, supports the designer's interactive interference while transforming an STG to equations, deriving, step-by-step, one equation after another and 'fitting' them into a given logic basis. Some routines for deriving equations directly from the NIV and NNV form (without using state graph) are being currently developed.

References

[1] S.B.Furber, P. Day, J.D. Garside, N.C. Paver, and J.V.Woods. AMULET1: A micropipeline ARM, In *Proceedings of VLSI'93*, Grenoble, France, Sept. 1993, Best Paper Award.

[2] M. Kishinevsky, A. Kondratyev, A. Taubin, and V. Varshavsky. *Concurrent Hardware: The Theory and Practice of Self-Timed Design*, Wiley and Sons, London, 1993.

[3] A. Kondratyev, J. Cortadella, M. Kishinevsky, E. Pastor, O. Roig, and A. Yakovlev. Checking Signal Transition Graph Implementability by Symbolic BDD Traversal, In *Proceedings of EDT Conference*, Paris, March 1995, IEEE Comp. Society Press, N.Y., pp. 325-332.

[4] L.Lavagno and A. Sangiovanni-Vincentelli. *Algorithms for Synthesis and Testing of Asynchronous Circuits*, Kluwer Academic Publishers, 1993.

[5] O. Roig, J. Cortadella, and E. Pastor. Hierarchical verification of speed-independent circuits, in *Proceedings 2nd Working Conference on Asynchronous Design Methodologies*, London, May 1995.

[6] T.G. Rokicki. *Representing and Modelling Digital Circuits*, Ph.D. thesis, Stanford University, 1993.

[7] E. Sentovich, K.J. Singh, L.Lavagno, C.Moon, R. Murgai, A. Saldanha, H. Savoj, P.R. Stephan, R. Brayton and A. Sangiovanni-Vincentelli, *SIS: A System for Sequential Circuit Synthesis*, Memo No. UCB/ERL/M92/41, Electronic Research Lab, University of California at Berkeley, May 1992.

[8] N.A.Starodoubtsev. Asynchronous processes and antitonic control circuits, *Soviet Journal of Computer and Systems Sciences(USA)*, English Translation of *Izvestiya Akdemii Nauk SSSR. Technicheskaya Kibernetika (USSR)*, 1985, Vol.23, No.2, pp.112-119 (Part I. Description language), No. 6, pp.81-87 (Part II. Basic properties), 1986, Vol.24, No.2, pp. 44-51 (Part III. Realization).

[9] N.A.Starodoubtsev. The Gate-Level Shell for Switching Circuitry and Its Usage for Synthesis and Verification. In: *Proceedings of International Design Automation Workshop*, Moscow, Russia, Summer 1992.

[10] J. Staunstrup. *A Formal Approach to Hardware Design*, Kluwer Academic Publishers, 1994.

[11] K. van Berkel, R. Burgess, J. Kessels, A. Peeters, M. Roncken, and F. Schalij. Asynchronous circuits for low power: a DCC error corrector, *IEEE Design and Test of Computers*, Summer 1994, pp. 22-32.

[12] K. van Berkel, J. Kessels, M. Roncken, R. Saeijs, and F. Schalij. The VLSI programming language Tangram and its translation into handshake circuits, In *Proceeedings of EDAC'91*, Brussels, March 1991, pp. 384-389.

[13] Ch.Ykman-Couvreur, B. Lin, H. De Man. *ASSASSIN: A Synthesis System for Asynchronous Control Circuits*, Tool Description and Manual, IMEC, 1995.