Characterizing Speed-independence of High-Level Designs [*]

Michael Kishinevsky Jørgen Staunstrup
Department of Computer Science
Technical University of Denmark
DK–2800 Lyngby, Denmark
email: {mik,jst}@id.dtu.dk

Abstract

This paper characterizes the speed-independence of high-level designs. *The characterization is a condition on the design description ensuring that the behavior of the design is independent of the speeds of its components. The behavior of a circuit is modeled as a transition system, that allows data types, and internal as well as external non-determinism. This makes it possible to verify the speed-independence of a design without providing an explicit realization of the environment. The verification can be done mechanically. A number of experimental designs have been verified, including a speed-independent RAM, a complex switch of a data path, various Muller C-elements, FIFO registers, and counters.*

1 Introduction

A circuit is speed-independent if its behavior does not depend on speeds of its components (gates). These circuits are very robust to parameter variations, such as supply voltage or temperature, and this may have significant practical advantages [8], for example, a potential reduction of power dissipation [13]. It is important to find a characterization of speed-independence that allows the designer to discover speed dependencies as early as possible in the design process. Such a characterization is presented in this paper as a sufficient condition on a high-level description of the circuit. The condition is formulated in such a way that the transition system can be checked in a modular way, i.e., by checking the design module by module.

Most characterizations of speed-independence assume that the circuit is *autonomous* which means that it is a self-contained circuit without external input. To check a component with external input (and output) an explicit environment is constructed and the combination of component and environment is then checked. Our characterization allows us to check a component in isolation, however, it is possible (and often necessary) to state assumptions about the environment. In the paper [7] it has been shown how the proposed check can be mechanized, however, in the present paper the emphasis is on the characterization itself. Circuit efficiency sometimes makes it necessary to compromise the speed-independence of a well-defined part of a circuit. Our characterization of speed-independence makes it possible to state some speed assumptions about well defined parts while still making it possible to check the rest of the design.

This paper is organized as follows. First, we present a short review of previously published characterizations of speed-independence. Section 3 describes the design language used in this paper for modeling circuits. Section 4 presents the characterization of speed-independence as a condition called persistency. Section 5 gives two examples of experimental designs where the proposed technique has been used to verify speed-independence. Finally, in Section 6 it is argued that the persistency condition is a sufficient condition for speed-independence.

2 Previous work

This section gives a brief overview of the characterizations of speed-independence that have been published previously.

2.1 Muller's model

In David Muller's theory [10] a logic gate is modeled by a logic function followed by an unbounded inertial delay element. Formally, a circuit consists of a set of boolean variables, $\{z_1, z_2, \ldots, z_n\}$ $(n \geq 1)$,

[*]This work has been supported by The Danish Technical Research Council and ACiD-WG (Esprit Basic Research Working Group 7225).

each of which has an associated boolean function: $f_i(z_1, z_2, \ldots, z_n)$, $1 \leq i \leq n$ and an initial state. A variable is either *stable*, if its value is equal to the new value computed by the corresponding boolean function, $z'_i = z_i$, or *excited* if $z'_i \neq z_i$. For example, an OR-gate with both inputs 0 and output 1 is excited. An excited variable is indicated by following its value with an "*". An excited variable can either perform the enabled transition or be *disabled* because the gate inputs change (e.g., if at least one input go to 1 before the output has been changed).

A circuit traverses states by changing values of excited variables either one at a time or in parallel. In other words, given an initial state or a set of initial states a circuit defines a state graph (a Transition Diagram [10]), that captures all states *reachable* from the initial states.

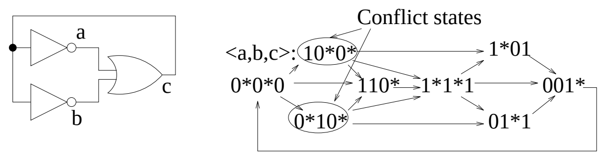

Figure 1: Speed-independent (by Muller) but not semimodular circuit.

Figure 1 shows a simple autonomous circuit with one OR-gate and two inverters.

A state $< z_1, z_2, \ldots, z_n >$ is a *conflict state* [5] if changing one excited variable disables another variable. Conflict states indicate that the local behavior of the components depends on their relative speed. A circuit with no reachable conflict states is called *semimodular*. Since there are two conflict states reachable from the initial state 0*0*0 in the state graph in Fig. 1, this circuit is not semimodular.

In Muller's original work a circuit is defined to be speed-independent if it has exactly one *final class* of behaviors that are reachable from the initial state. The final class is a closed set of reachable states inside which each excited variable must change its value. This definition allows for disabling of excited variables as exemplified in Fig. 1. Here all states form only one final class, since all states are reachable from any other state. Therefore, this is an example of a circuit meeting Muller's definition of speed-independence which is not semimodular.

In later work, the notion of semimodularity is often used as a characterization of speed-independence, because:

- semimodularity is easier to analyze than Muller's definition of speed-independence,
- semimodularity is robust with respect to the delay model used for gates [1].

2.2 Speed-independence in trace theory

Trace theory is based on characterizing properties of traces obtained by computations of the circuit [2, 3, 4, 14]. The designer is assumed to describe a design with trace commands [3]. Figure 2.a shows the circuit from Fig. 1. The circuit consists of two components: an initiated fork (implemented by a wire fork and two inverters) and an OR-gate.

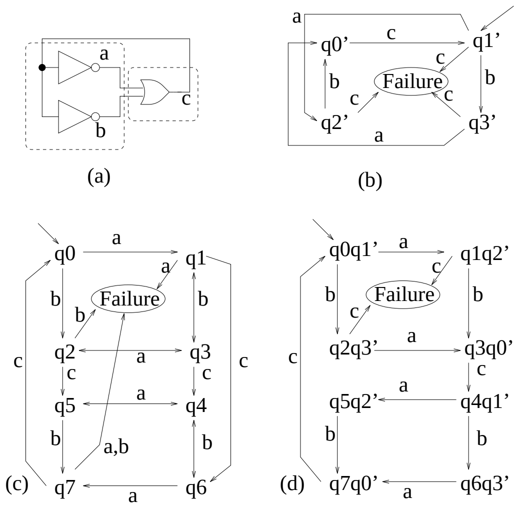

Figure 2: Design with computation interference (a), and state graphs for initiated fork (b), OR (c), and OR || Fork (d).

Regular trace structures can only represent regular sets. Therefore, an automaton can be constructed for each of the components, which will accept the same regular set. A state graph for the initiated fork is given in Fig. 2.b and for the OR-gate in Fig. 2.c. In circuits that have components with symmetric operations for rising and falling signals, one abstract state might correspond to several binary signal values. This is, for example, the case for the initiated fork. The behavior of the OR-gate is not symmetric. Therefore, a state graph for the OR-gate (Fig. 2.c) has eight states, $q0 - q7$, which correspond to the eight possible binary states of a two input OR-gate. The trace command specification of the OR-gate requires internal state variables and is similar to the structure of the state graph from Fig. 2.c. This indicates a drawback of trace commands as a specification language for circuits: the size of the specification grows exponentially even for simple gates such as OR, AND, NOR, NAND.

A circuit cannot control the transitions on its inputs, since these are made by the environment. Therefore, any input transition can occur in any state of the state graph (*receptiveness*) [2]. However, some of the traces, called *failure traces*, may induce hazards. After an input transition on a has occurred in the state $q0$ the OR-gate goes to state $q1$, and it is ready to produce an output on c and go to state $q6$. This corresponds to the state 10*0* (in Fig. 1). However, if the same input a changes in state $q1$, this causes a transition to the *failure* state. This corresponds to disabling of the OR-gate by changing the input a back to 0. Such a failure trace is called *computation interference* [14] or *choking* [2].

To analyze the behavior of the complete circuit the state graphs of the components are composed. In the example states in the composed state graph (Fig. 2.d) are labeled by pairs of states of the OR-gate and the initiated fork, e.g., the initial state of the composed graph is $< q0, q1' >$. A transition between states labeled with a common symbol of several components may occur only if all circuit components that have this symbol in their alphabet can perform this transition. In the state graph for the complete circuit (Fig. 2.d), the failure state is reachable from the states $< q1, q2' >$ and $< q2, q3' >$ that correspond to the conflict states 10*0* and 0*10* in Fig 1. This indicates that *computation interference in a trace structure corresponds to violation of semimodularity* in the Muller model.

2.3 Tools for speed-independence

There are several model-checking tools for checking speed-independence based both on the Muller model of circuits and the trace model [1, 2, 4, 5]. A comparison of these tools is given in [5]. The rest of this paper is devoted to a characterization of speed-independence making it possible to check high-level design descriptions. This characterization gives some new possibilities compared with the techniques mentioned above:

- non-autonomous designs are handled without giving an explicit environment. This makes it possible to check large designs piece by piece in a modular fashion,
- higher level designs with non-binary data types can be handled,
- verification of other (safety) properties of a design can be done using exactly the same approach and the same tools. Hence, speed-independence does not require separate and specialized tools,
- no distinction is made between control and data dominated designs; both can be handled with the same approach.

3 High-level design descriptions

This section describes how to model circuits as formal transition systems using the design language Synchronized Transitions [12]. Such transition systems consist of a set of transitions and a set of state variables (both are fixed and do not change during a computation).

3.1 Design descriptions

A *transition*, t, describes a component of a circuit and has the form $\ll C^t \;\text{->}\; z^t := E^t \gg$, where C^t – is a predicate called the *precondition*, z^t is a state variable, and E^t is an expression, that has a unique value in any state. As an example, consider a C-element, this is described as follows: $\ll a = b \;\text{->}\; c := a \gg$. In this example, a, b, and c are boolean state variables, and whenever $a = b$, it is possible to assign the value of a to c. If $a \neq b$, then c keeps its current value.

A circuit with many components (operating in parallel) is described by composing a number of such transitions (one for each component). The oscillator from Figure 1 is described as follows:

$$\ll a := \neg\, c \gg \;||\; \ll b := \neg\, c \gg \;|| \\ \ll c := a \;\vee\; b \gg$$

When a precondition is the constant *TRUE*, it can be omitted as illustrated by the three transitions of this design. State variables are introduced by a variable declaration that defines a type of the variable and this type determines the set of values that the state variable can take. By using other types than boolean, e.g., integers or arrays, it is possible to model computations on composite values. An integer multiplication is, for example, described as follows: $\ll z := \; s * t \gg$ where s, t, and z are state variables of type integer. Further details on Synchronized Transitions are given in the book [12].

3.2 Terminology

This section defines a number of concepts that are used to characterize speed-independence.

The transition t is *enabled* in a state s if C^t is satisfied in s, and t is *active* if it is enabled and $z^t \neq E^t$ in s. The predicate $active^t$ is: $active^t \;\equiv\; C^t \wedge (z^t \neq E^t)$.

When this predicate is explicitly applied in a particular state, s, it is written as: $active^t(s)$.

A transition defines a set of ordered pairs of states; for each such pair the first element is called the *pre-state* and the second the *post-state.* For every transition, t, there is a corresponding predicate $t(pre, post)$ defined on the pairs of states. This predicate is true if and only if the transition t can change the state from *pre* to *post.* $z^t.post$ denotes the value of z^t in the post-state, and similarly $E^t.pre$ is the value of an expression E in the pre-state, so $z^t.post = E^t.pre$.

The *write set*, W^t, of a transition, t, is the set of state variables appearing on the left-hand side of assignments. Similarly, the *read set*, R^t, of a transition is the set of state variables that appear in the precondition and on the right-hand side of the assignment.

SYNCHRONIZED TRANSITIONS has syntactic constructs for encapsulating parts of a design into cells which may have internal state variables that are invisible outside the cell. In this paper, the following simplified distinction is made between state variables. If the variable z does not belong to any of the write sets, i.e., no transition can change its value, then it is called an *external* variable. Otherwise z is called an *internal* variable. External variables correspond to the input signals of a design. Some of the internal variables serve as outputs.

The following definitions describe a restricted set of design descriptions, called well-behaved designs, for which it is possible to ensure a one-to-one correspondence between the design description and a circuit realization.

Definition 1 *The transitions* $t_1, t_2, \ldots t_n$ *meet the* exclusive write *condition if and only if:*

$$\forall i, j \in [1..n] : W^{t_i} \cap W^{t_j} \neq \emptyset \Rightarrow \neg(C^{t_i} \wedge C^{t_j})$$

This condition ensures that there are no states where different enabled transitions can assign to the same state variable.

Definition 2 *The transitions* $t_1, t_2, \ldots t_n$ *meet the* unique write *condition if and only if for each state variable z and for each value v in the domain of state variable z there is a unique transition that can assign value v to the state variable z.*

This condition ensures that for each value that a state variable may get, it is possible to identify a unique transition assigning that value.

Definition 3 *A design is called* well-behaved *if it obeys the exclusive write and unique write conditions.*

It can be shown that it is possible to describe any asynchronous circuit as a well-behaved design.

3.3 Operational model

The computation of a design can be modeled as repeated non-deterministic selection and execution of an enabled transitions. Transitions are executed: *one at a time,* i.e., as an indivisible operation, *repeatedly,* each time one has been executed, it is immediately ready to be selected again, *independently,* of the order they appear in the design description. There is no upper bound on when a transition is selected.

A design defines a set of computations that are sequences of states, called *trajectories*. The formal definition of trajectories is given in Section 6.2.

3.4 Invariants and protocols

Invariants and protocols describes properties of a design that can be verified formally.

Invariants: are predicates over the state variables, defining restrictions (subsets) on the state space.

Protocols: are predicates on pairs of states, *pre*, *post*, defining restrictions on the allowable transitions between states.

For example, the following invariant states that x and y cannot be true simultaneously (mutual exclusion).

$\neg\ (x \wedge y)$

The following is an example of a protocol stating that whenever x changes, it gets the value of y.

$x.pre \neq x.post \Rightarrow\ x.post{=}y.pre$

3.5 Environments

In general, the computation of a design depends on the behavior of its environment. Protocols and invariants are used as implicit specifications of environments expressing constraints on the state space and possible transitions changing *external* state variables.

Example: a pipeline latch. Consider a pipeline latch, it is described as a design with four state variables: two booleans, ai, ao, to model the binary acknowledgment signals and two duals, Di, Do, to model a one bit data path. The domain for the duals contains three possible values $\{E, T, F\}$ ("empty", "true", and "false"). The value E is used to reset the latch before it can adopt the next valid data value, T or F. The variables ao (the output acknowledgment) and Di (input data) are external. Figure 3.a shows the structure of the latch; figure 3.b shows one possible gate-level realization based on two C-elements and one NOR-gate.

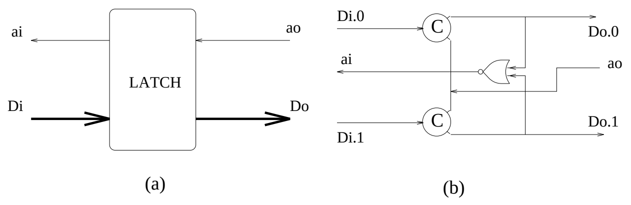

Figure 3: A structure of the latch (a) and its gate-level implementation (b)

Let *empty* be a predicate which returns *TRUE* when the value of the dual parameter is equal to *E*. Then the latch is described as follows:

$$\ll ai{:=}\ empty(Do) \gg \ || \\ \ll ao \neq empty(Di)\ \ {-}{>}\ \ Do{:=}\ Di \gg$$

The duals Di and Do alternate between the value E and T or F, and they must only change after the previous change has been acknowledged. These assumptions are expressed with the following protocol on Di and ao:

$$\begin{aligned} P_E \equiv\ & (Di.pre{\neq}Di.post \Rightarrow \\ & \quad (ai.post{\neq}empty(Di.post)) \wedge \\ & \quad (ai.post{=}empty(Di.pre))) \bigwedge \\ & (ao.pre{\neq}ao.post{\Rightarrow}(ao.post{=}empty(Do.post))) \end{aligned}$$

The protocol constrains any change of Di to start in a pre-state where $ai.post{=}empty(Di.pre)$. This prevents Di from changing directly from one non-empty value to another. Note that a latch for a wider data path is specified by substituting another type instead of dual.

End of example

3.6 Internal non-determinism

Protocols and invariants are also used to specify components with internal non-determinism, for example an arbiter. As an example, consider a simple arbiter serving two clients. Each client indicates a request by making the state variable Req_1 true, the arbiter gives an acknowledgment by making Ack_1 true. The behavior of the two-input arbiter is defined implicitly by the invariant $\neg(Ack_1 \wedge Ack_2)$ and the protocol:

$$\begin{aligned} & (Ack_1.pre \neq Ack_1.post \Rightarrow \\ & \neg Ack_2.pre \wedge \neg Ack_2.post \wedge Ack_1.post = Req_1.pre) \bigwedge \\ & (Ack_2.pre \neq Ack_2.post \Rightarrow \\ & \neg Ack_1.pre \wedge \neg Ack_1.post \wedge Ack_2.post = Req_2.pre) \end{aligned}$$

The *internal* behavior of the arbiter is not a subject of verification (it cannot be verified by logic means anyway), but a cooperative behavior of the arbiter with other components is verified. This allows us to check the speed-independence of designs with internal non-determinism.

3.7 Design

A design, D, is a five-tuple $< Z, T, P_E, U, I >$, where Z is a finite set of state variables; T is a finite set of transitions; P_E is an external protocol, restricting possible transitions of external state variables, U is a set of initial states, and I is an invariant.

Formally, the protocol and invariant of a design constrains both the internal and external state variables. However, in practice it can be useful to distinguish the external constraints from the internal. The external serves as an implicit characterization of the environment and this is usually needed to carry out the verification. On the other hand, the internal constraints can often be derived automatically. In [7] it is described how model-checking is used for automatically deriving an invariant defining the reachable states.

Example: the pipeline latch (continued). The invariant for the pipeline latch with the external protocol P_E is characterized by the following expression:

$$\begin{aligned} I \equiv\ & (Di = Do) \vee (ai \wedge empty(Do)) \vee \\ & (\neg\ ai \wedge empty(Di)) \end{aligned}$$

End of example

For simple designs, it is possible to derive the invariant manually, but for more challenging designs this is often too laborious and automatic derivation is therefore useful.

4 The persistency condition

This section presents a characterization of the speed-independent designs. It is formulated as a condition on a design; when it is met, the design allows for a speed-independent circuit realization. In Section 6 it is argued that the condition is sound, i.e., that it ensures that a computation is independent of the speed of its components. The condition is used for checking designs and this has influenced the formulation of the condition. It is expressed as a protocol which makes it possible to use existing verification techniques and tools to check the condition.

The protocol $Persistent^t(pre, post)$ defines the constraint that transition t stays active, providing the same post value for the write variable, while other transitions occur. It is defined as follows:

$$Persistent^t(pre, post) \equiv$$
$$Active^t(pre) \Rightarrow (Active^t(post) \wedge E^t.pre \;=\; E^t.post)$$

The persistency protocol generalizes the notion of a conflict state from the Muller model (see Section 2.1) for high-level designs with variables of any finite type and with internal and external non-determinism.

Example: The pipeline latch (continued). The persistency protocol for the last transition of the latch from Fig. 3 is:

$$(ao.pre \neq empty(Di.pre)) \wedge (Do.pre \neq Di.pre) \Rightarrow$$
$$(ao.post \neq empty(Di.post)) \wedge (Do.post \neq Di.post)$$
$$\wedge\ (Di.pre = Di.post)$$

End of example

A design is persistent if the persistency protocol is met for all transitions of the design, i.e., if any state change by an internal transition or in the environment meets the persistency protocols of all transitions

Definition 4 *Let D be a design $< Z, T, P_E, U, I >$. Then D satisfies the* persistency *condition, if the following can be shown:*

(1) for all pairs of transitions t_1, t_2 in T, $t_1 \neq t_2$:
$t_1(pre, post) \wedge I(pre) \Rightarrow Persistent^{t_2}(pre, post)$

(2) for any transition t in T:
$P_E(pre, post) \wedge I(pre) \Rightarrow Persistent^t(pre, post)$.

When a design meets the persistency condition, it is ensured that no active variable is disabled by the state changes of other transitions or by the state changes of the external variables.

Example: an oscillator (continued). To illustrate a non-persistent design consider the oscillator Fig. 1 and its description in Section 3. In the state $a, b, y = FALSE, FALSE, FALSE$ both the first and the second transitions are active. If the first transition changes c to $TRUE$, then the second is no longer active; therefore, the implication in the persistency protocol does not hold, and hence the design is not persistent.

4.1 Mechanizing the check

The paper [7] describes tools for mechanically checking the persistency condition. They consists of:

- a translator for transforming design descriptions into a list of proof obligations corresponding to the persistency condition;
- a tool for generating reachability invariants;
- a theorem prover (the LARCH PROVER) that is used to verify the proof obligations.

Note that the persistency condition yields a separate implication for each transition. Hence, the verification is broken into a number of independent steps.

Example: the pipeline latch (continued). The invariant, called I in Section 3.7, can be used to verify that the pipeline latch meets the persistency condition. The external protocol, PE, for the latch was defined in Section 3.5.

For each (of the two) transitions of the design, it must be shown that it satisfies the persistency protocol of the other transitions (in this case there is only one), and the external protocol PE

$$I(pre) \wedge t_i(pre,\ post) \Rightarrow Persistent^{t_j}(pre,\ post)$$
$$I(pre) \wedge P_E(pre,\ post) \Rightarrow Persistent^{t_j}(pre,\ post)$$

Where $i, j \in 1, 2 \wedge i \neq j$. Given the design description, the tools mentioned above, generates similar implications and verifies them which shows that this is a speed-independent design.

5 Applications

This section describes two particular designs: (1) a switch used in the data-path of a multiplier – this illustrates the use of high-level designs with variables of non-boolean type, and (2) a self-timed RAM design – this illustrates how to do a partial check of a design with a delay assumption about a well defined part.

5.1 A switch of a data-path

The asymmetric switch shown in Figure 4 is used in a speed-independent vector multiplier design [11]. This switch either lets both data signals pass through, or it crosses one of them over and ignores the other. All signals in this design follows a four-phase protocol. The data lines, *InA, InB, OutA*, and *OutB*, are part of a dual-rail encoded data paths (of arbitrary width) and one single-rail acknowledgement signal to (for *InA, InB*) or from (for *OutA, OutB*) the environment. The control input, *Ctl*, also follows a four-phase protocol, and it is a dual-rail input signal. Finally, there is a boolean acknowledgement signal to the environment.

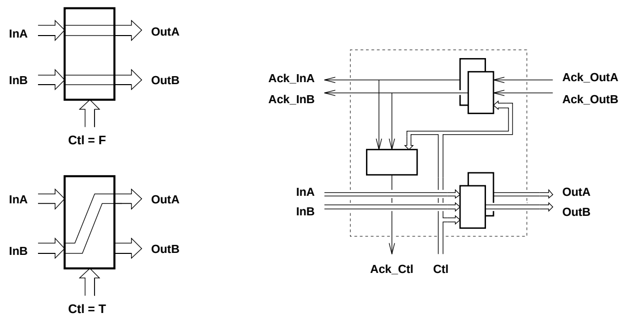

Figure 4: Asymmetric data-path switch

Details of this design are given in [11]. Here, only a part of the design description is shown, it specifies the behavior of the data paths for transferring one dual-rail encoded word (word width n). Each of the dual-word variables is implemented by $2n$ wires and can take $2^n + 1$ different values: one empty value, E, and 2^n valid combinations. Note that we do not need to explicitly enumerate states corresponding to all valid code combinations. Instead, a predicate $\neg$*(empty(InA))* is used to characterize all valid values of the variable *InA*.

$$\ll (Ctl{=}T) \neq empty(InB)\ \ {-}{>}\ \ a1{:=}\ InB \gg \ ||$$
$$\ll (Ctl{=}F) \neq empty(InA)\ \ {-}{>}\ \ a0{:=}\ InA \gg \ ||$$
$$\ll (Ctl{=}F) \neq empty(InB)\ \ {-}{>}\ \ OutB{:=}\ InB \gg \ ||$$
$$\ll OutA{:=}\ IF\ a0{=}E\ THEN\ a1\ ELSE\ a0 \gg$$

The switch is apparently very simple, but it was quite difficult to find a correct speed-independent design. The formal verification revealed several mistakes in designs that were believed to be correct and where careful simulations had not uncovered any errors.

5.2 A RAM cell

This section describes the design of a RAM cell, and it is shown how to do a partial check of speed-independence. It turns out that a small well-defined part of the RAM cell is not speed-independent. However, this part can be excluded from the check, and the rest of the design can be verified.

The major difficulty in designing a speed-independent RAM is in the implementation of the write operation. It is non-trivial to organize a completion detection after a new value has been written into the memory cell. Figure 5.a shows a design (for the write operation only) of a self-timed memory that has been made by Lars Nielsen, DTU, in 1993.

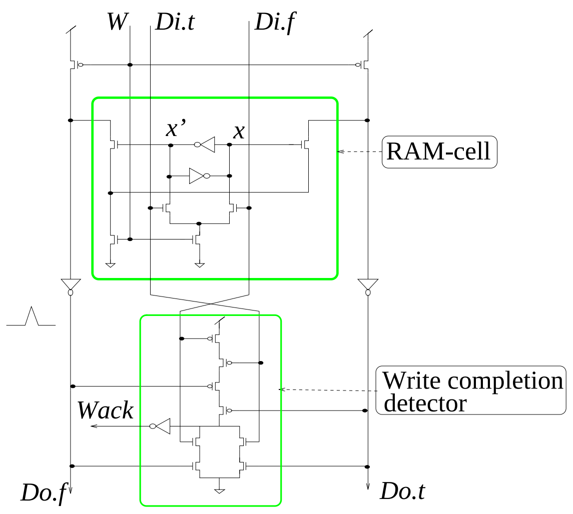

Figure 5: Self-timed RAM (write operation)

Each static memory cell contains 10 transistors, and each column of the memory array includes one write completion detector. The memory cell is described as follows:

$$t_{x'} : \ll x' := \neg\ ((Dit \wedge W) \vee x\ ) \gg \ ||$$
$$t_{x} : \ll x := \neg\ ((Dif \wedge W) \vee x') \gg \ ||$$
$$t^1_{Dof} : \ll x' \wedge W\ \ {-}{>}\ \ Dof{:=}\ TRUE \gg \ ||$$
$$t^2_{Dof} : \ll \neg\ W\ \ {-}{>}\ \ Dof\ {:=}FALSE \gg \ ||$$
$$t^1_{Dot} : \ll x \wedge W\ \ {-}{>}\ \ Dot{:=}\ TRUE \gg ||$$
$$t^2_{Dot} : \ll \neg\ W\ \ {-}{>}\ \ Dot\ {:=}FALSE \gg$$

The completion detector is specified as follows:

$$t^1_{Wack} : \ll Dit \wedge Dot\ \vee\ Dif \wedge Dof\ {-}{>}\ Wack{:=}TRUE \gg ||$$
$$t^2_{Wack} : \ll \neg(Dit \vee Dot \vee Dif \vee Dof)\ \ {-}{>}\ Wack{:=}False \gg$$

A check of the design shows that the transitions t^1_{Dof} and t^1_{Dot} do not meet persistency, and hence, this design is not speed-independent. Moreover, the mutual exclusion condition for variables *Dot* and *Dof* is not met, and therefore two '1' values can appear at the output data lines. If the RAM-cell stores a '0' value, i.e., $x = 0$ and $x' = 1$, and the input data value is '1' (*Dit=1* and *Dif=0*), then immediately after arrival of the write control signal *W* two transitions start: writing a '1' value into the cell, and driving the output data line *Dof* to become 1. Depending on the relative speeds of these processes either a short voltage spike appears at the line *Dof* (non-persistency) or this line will hold a '1' value until the next cycle (where

Dot may take a '1' value which implies that mutual exclusion is violated).

However, the RAM cell operates correctly if it is assumed, that the write control signal W always goes high with a delay of at least τ after *Dit, Dif* gets a valid value $(0,1$ or $1,0)$, where τ is bigger than the delay of the memory cell. Such an assumption is typical for some asynchronous and self-timed design styles, e.g., micro-pipelines and systems with a bundled data protocol. For such designs, it is possible to do a *partial* check for speed-independence. For example, in the RAM cell design, we can exclude the two transitions, t^1_{Dot} and t^1_{Dof}, from the persistency check and verify that the rest of the design is speed-independent.

6 Soundness of the characterization

This section shows how to formulate the intuitive notion of speed-independence and relates the class of persistent designs to this definition.

6.1 Delayed designs

The intuitive notion of gate delays is modeled by the notion of a *delayed* design.

Definition 5 *Let D be a design with transitions T and external protocol P_E and let z be an arbitrary state variable of a design. A design with a delayed variable z is constructed in three steps:*

(1) One transition is added to the design $\ll z^d := z \gg$ where z^d is a new variable.

(2) In all *transitions $t_i \in T$ where $z \in R^{t_i}$* all *occurrences of the state variable z are replaced by z^d.*

(3) Occurrences of $z.pre$ in the external predicate $P_E(pre, post)$ are replaced by $z^d.pre$.

Applying this definition iteratively, one gets a version of the design with multiple delays including multiple delays of a single state variable z.

In the underlying circuit, the delay of z corresponds to inserting a delay element *before the fork* of a wire delivering the value of z to other components and to the environment. If z is an internal variable, then the delay is inserted before any forks of the internal wire. If z is an external variable, then the delay is inserted before the fork of the input wire.

6.2 Equivalence of designs

A design defines a set of computations that are *trajectories of states*, $S_0, S_1, \ldots$, where $S_0 \in U$ is an initial state, each state S_i satisfies the invariant $I(S_i)$ and for each pair of states, S_i, S_{i+1}, one of the following conditions is met:

- S_{i+1} differs from S_i by the value of an internal variable, z, and there is a transition, t, such that $z \in R^t$ and $t(S_i, S_{i+1})$, i.e., S_i is a pre-state of t, and S_{i+1} is a post-state of t or
- S_{i+1} differs from S_i by the value of an external variable and $P_E(S_i, S_{i+1})$ is satisfied.

The set of all trajectories of a design D is denoted $Tr(D)$. If S_{i+1} differs from S_i by the value of a variable x, and if x has the value v in the state S_{i+1}, then we will say that variable x can perform an assignment $x := v$ in the state S_i and denote it $S_i \xrightarrow{x:=v} S_{i+1}$.

A *projection* operator on trajectories is defined as follows:

Definition 6 *Let $Tr(D)$ be the trajectories of design D and $Z' \subset Z$ be a subset of the variables.*

(1) a projection of a state S onto Z', $S \downarrow Z'$, is a state S' derived from S by deleting all components corresponding to the variables $z \in Z - Z'$;

(2) a projection of a trajectory is constructed in two steps: first, all states of the trajectory are projected and then equal adjacent state projections are collapsed;

(3) a projection of the trajectories on Z', $Tr(D) \downarrow Z'$, is a set of all trajectory projections $\{s \downarrow Z' | s \in Tr(D)\}$.

The projection is used to defined the equivalence of a design and its delayed version.

Definition 7 *Design D with a set of state variables Z and its delayed version $D^d_{Z'}$, where $Z' \subset Z$, are* observation equivalent *if $Tr(D) = Tr(D^d_{Z'}) \downarrow Z$.*

6.3 Persistency of the environment

The persistency condition encourages an approach where a component and its environment are checked independently. If the component and the environment are both specified as transition systems then the persistency condition can be used on both. However, if the environment is specified by other means, it should still behave persistently. This section defines a restriction called *external persistency.* If the external persistency is met by a design, then the design is speed-independent for any environment satisfying the external protocol of the design. It must be stressed that transition systems satisfying the persistency condition automatically satisfies the restriction; it is only relevant for environments specified by other means.

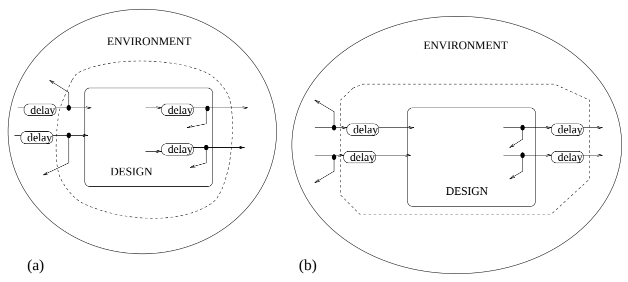

Figure 6: A speed-independent (a), and delay-insensitive (b) composition of a design and its environment.

The *Foam Rubber Wrapper* property [9] is often used for delay-insensitive circuits. It states that if arbitrary delays are attached to the input and output lines of the implemented system, the new interface created must have the same behavior as the originally specified (Fig. 6.b).

A corresponding property for a speed-independent environment would be to attach arbitrary delays to the input and output lines *before* wire forks such that the environment (in case of input lines) or the design (in case of output lines) observes delayed signals (Fig. 6.a). This requirement to a speed-independent environment can be captured by the *external persistency* condition. The external persistency is weaker than persistency, since it allows non-deterministic behavior of the environment.

Similar to the persistency condition the external persistency condition consists of two requirements. These constrain the behavior of external variables. Intuitively, the first requirement states that if two external variables, are *concurrently active* in a state of the design then they can change their values in any order. The second requirement states that transitions of internal variables cannot disable external variables.

Definition 8 *The design D meets the external persistency condition if two requirements hold:*

1. *If two external variables, z and x can perform the assignments $z := v$ and $x := w$ according to the external protocol P_E in a reachable state S_1 such that $S_1 \xrightarrow{z:=v} S_2$, $S_1 \xrightarrow{x:=w} S_3$ and furthermore $S_2 \xrightarrow{x:=w} S_4$, i.e., the assignment $x := w$ is still possible in S_2 after z has changed its value, then z can perform the same assignment $z := v$ in state S_3: $S_3 \xrightarrow{z:=v} S_4$.*

2. *If some external variable z can perform an assignment $z := v$ according to the external protocol P_E in a reachable state S_1 and arbitrary transition t_i is active in S_1 such that $t_i(S_1, S_2)$, then z can perform the same assignment $z := v$ in the state S_2.*

It is important to notice that verification of the complete design does *not* require a check for the external persistency condition in those cases where all modules of the design are expressed as transition systems.

6.4 Soundness of persistency

This section defines the notion of a speed-independent design and states a theorem relating the persistency condition to speed-independence.

Definition 9 *A design in D is* speed-independent, *if any delayed version of design D is observation equivalent to D.*

Although this definition requires all possible delayed versions to be observation equivalent to the original design, it is not necessary to compare all multiple delayed versions. It can be shown that only single delayed versions need to be considered. Furthermore, instead of checking an observation equivalence one can simply check the persistency condition.

Theorem 1 *If a well-behaved design satisfies the persistency condition and its environment satisfies the external persistency condition, then the design is speed-independent.*

The proof of this theorem is given in [6].

A sketch of the proof. The theorem is first proved (by contradiction) for a case when *one* variable is delayed. Let D_z^d be a delayed version that is not observation equivalent to D, although both the persistency and the external persistency hold. Let s be the shortest possible trajectory in the delayed design D_z^d that has no equivalent projection in the original design D. It can always be represented in the form $s = r, S_0 \xrightarrow{z} q, S_1 \xrightarrow{y} S_2$, where r, q are trajectories (both r and q might be empty), S_0, S_1 and S_2 are states of the delayed design, z is a delayed variable, y is another variable of the design, and $S_0 \xrightarrow{z} q$ is the last assignment to z in s.

Assume that q is not empty. No transitions in q can read z in the delayed design. Hence, instead of the trajectory s one can always consider another trajectory of the same length, which is obtained from s by swapping the last assignment to z and all the others assignments to the variables that occurs along the trajectory q. Therefore, we can restrict consideration to trajectories where q is empty: $s = r, S_0 \xrightarrow{z} S_1 \xrightarrow{y} S_2$.

Four cases are possible: (1) z, y are internal variables, (2) z is external and y is internal, (3) both z and y are external, and (4) z is internal and y is external.

Let us consider the first case. Since by assumption z and y are internal variables, then there is a unique transition $t_1 : \ll z := E_{t_1} \gg$ that is active in S_0 and there is a unique transition $t_2 : \ll y := E_{t_2} \gg$ that is active in S_1. By construction of a delayed design the values of R^{t_2} in the state S_1 for design D^d_z are exactly the same as values of R^{t_2} in the state $S_0 \downarrow Z$ for design D. Therefore, for design D, t_2 is active in the state $S_0 \downarrow Z$ but it is either not active in $S_1 \downarrow Z$ or $E_{t_2}.(S_1 \downarrow Z) \neq E_{t_2}.(S_0 \downarrow Z)$. By definition the first requirement of the persistency condition for t_2 is not satisfied. We have reached a contradiction.

Similarly, for the second case the contradiction is reached with the second requirement of the persistency condition for the transition t_2, for the third and forth cases a contradiction is found with the first and the second requirements of the external persistency condition.

The case with more than one variable delayed is reduced to the case with one variable to be delayed.

$\square$

In [6] it is also shown that the persistency and external persistency conditions are necessary for a well-behaved design to be speed-independent for all environments satisfying the external protocol.

7 Conclusion

This paper has presented a sufficient condition for the speed-independence of a high-level design. The description of such high-level designs allow variables of any finite type and hierarchical structure with both external (input choice) and internal (arbiters) non-determinism. The formulation of the condition was related to other characterizations of speed-independence. The major difference of the condition presented here is the emphasis on independent verification of separate components/modules of a design.

Acknowledgements

We are grateful to Alex Kondratyev and Alexander Taubin for numerous discussions on speed-independence, to Michael Mendler for discussions of this paper, and to Niels Maretti for his work on automatic generation of reachability invariants.

References

[1] P.A. Beerel and T.H.-Y. Meng. Semi-modularity and testability of speed-independent circuits. *Integration, the VLSI journal*, 13(3):301–322, September 1992.

[2] D.L. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits.* The MIT Press, Cambridge, Mass., 1988.

[3] Jo C. Ebergen. A formal approach to designing delay-insensitive circuits. *Distributed Computing*, 5(3):107–119, 1991.

[4] Jo C. Ebergen and S. Gingras. A verifier for network decompositions of command-based specifications. In *Proc. Hawaii International Conf. System Sciences*, pages 310–318. IEEE Computer Society Press, 1993.

[5] M. Kishinevsky, A. Kondratyev, A. Taubin, and V. Varshavsky. Analysis and identification of speed-independent circuits on an event model. *Formal Methods in System Design*, 4(1):33–75, 1994.

[6] M. Kishinevsky and J. Staunstrup. Checking speed-independence of high-level designs (full version). Internal Technical Report, Department of Computer Science, Technical University of Denmark. May 1994.

[7] M. Kishinevsky and J. Staunstrup. Mechanized verification of speed-independence. In *Proceedings of the 2nd Workshop on Theorem Provers in Circuit Design*, Germany, September 1994.

[8] A. Martin, S. Burns, T. Lee, D. Borkovic, and P. Hazewindus. The first asynchronous microprocessor: the test results. *Computer Architecture News*, 17(4):95–110, June 1989.

[9] C.E. Molnar, T.P. Fang, and F.U. Rosenberger. Synthesis of delay-insensitive modules. In H. Fuchs, editor, *Proceedings of the 1985 Chapel Hill Conference on VLSI.* Computer Science Press, 1985.

[10] D. E. Muller and W. C. Bartky. A theory of asynchronous circuits. In *Annals of Computing Laboratory of Harvard University*, pages 204–243, 1959.

[11] J. Sparsø and J. Staunstrup. Delay-insensitive multi-ring structures. *INTEGRATION, the VLSI Journal*, 15(3), 1993.

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

[13] K. van Berkel, R. Burgess, J. Kessels, A. Peeters, M. Roncken, and F. Schalij. A Fully-Asynchronous Low-Power Error Corrector for the DCC Player. In *ISSCC 1994 Digest of Technical Papers*, volume 37, pages 88–89, San Francisco, 1994.

[14] J. L. A. van de Snepscheut. *Trace Theory and VLSI Design*, volume 200 of *Lecture Notes in Computer Science.* Springer Verlag, Berlin, 1985.