Mechanized Verification of Speed-independence

Michael Kishinevsky and Jørgen Staunstrup

Department of Computer Science, Tech. Univ. of Denmark,
DK-2800 Lyngby, Denmark.
e-mail: {mik,jst}@id.dth.dk

Abstract. Speed-independence is a property of a circuit ensuring correct operating regardless of the magnitude of delays in all its gates. In this paper, circuits are modeled by formal transition systems, and speed-independence is characterized by state predicates expressing constraints on the transition system. This makes it possible to define a formal condition corresponding to speed-independence, and to mechanically verify that a given transition system satisfies the condition. The condition is formulated in such a way that the transition system, and hence also the circuit design, can be checked in a modular way, i.e., by checking the circuit design module by module. This means that large designs can be checked in smaller pieces and without providing an explicit circuit realization of the environment.
A number of designs have been verified using the approach described here, including a speed-independent RAM cell, a complex switch of a data-path, and a number of standard components such as counters, FIFO registers, and various Muller C-elements.

1 Introduction

The correct operation of a speed-independent circuit does not depend on the delays of its components (gates). Such circuits are very robust to data and parameter variations. This may have significant practical advantages [11, 14], for example, a potential reduction of power dissipation [18]. However, to realize a design by a speed-independent circuit, the design must meet some constraints excluding behavior that depends on timing details of the components. Hence, a designer must not violate these constraints. There are several ways to achieve this, one would be to follow a "correct by construction" approach [11]; in this paper another alternative is explored, using mechanical tools to check that a high-level description of the designs behavior meets certain conditions (ensuring speed-independence). The following standard example is used throughout the paper to introduce and motivate the approach.

Example: Modulo-N Counter. The modulo-N counter with constant response time is a simple, yet interesting, example of a speed-independent design [5]. To save space, it is assumed that N is a power of two, and therefore the counter is called a modulo-2^n counter. The counter has one input, a, and two outputs p and q. Every signal change on the input a is acknowledged by a signal change of either p or q. The first $2^n - 1$ up-going changes on a are acknowledged by up-going changes on p and the last, 2^n-th, by an up-going change on q. The same with down-going changes. The counter cell is composed of a toggle element, a pipeline latch, and an OR-gate. The design used in this paper uses a four-phase protocol and was done by Christian D. Nielsen [15], it has many similarities with the two-phase design described in [5].

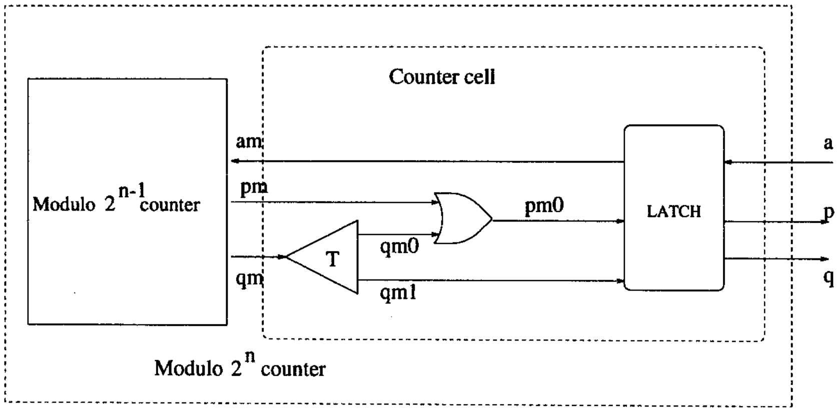

Fig. 1. A Diagram of the Modulo-2^n Counter

End of example

This paper introduces a technique for checking that a high-level description of a design, such as the modulo-N counter, allows for a speed-independent circuit realization. By using a high-level description, it becomes possible to check the design early in the design process. The behavior of a circuit defined by a high-level description is modeled as a transition system, and the conditions ensuring speed-independence are expressed as predicates on the state space of the transition system. The conditions are formulated in such a way that the transition system, and hence also the circuit design, can be checked in a modular way, i.e., by checking the circuit design module by module. The modulo-2^n counter consists of a toggle module (called T in Fig. 1), a few gates, and a modulo-2^{n-1} counter module (which itself may have further submodules). The basis of the recursive specification is a modulo-1 counter which simply connects the input a to the output q by a wire, and assigns the constant value false to the other output p.

The hierarchical nature of a design is exploited in the speed-independence check by treating modules as black boxes where the internal details are hidden. As a consequence, it is also possible to check a particular module, such as the toggle, without providing an explicit circuit realization of the environment, i.e., to check non-autonomous designs. This means that large designs can be checked in smaller pieces, and that designs can be checked without providing an explicit circuit model of the environment.

There are already several verification tools for checking speed-independence based on model-checking. For example, in [1, 8] different methods have been presented based on a direct construction of the circuits state space or its essential subspaces. In [3, 4] speed-independence is verified as an absence of choking (or computational interference) in a trace-based specification of a circuit. These approaches use special purpose tools aimed at verifying a limited set of properties. The work described in this paper differs in several respects. Most importantly with respect to the level of the circuit specification. It is possible to check designs with composite data-types, e.g., n-bit words, hierarchy (which is maintained in the verification), and non-autonomous parameterized designs. Another interesting difference is that the tools used to check for speed-independence are general purpose and not constructed specifically for speed-independence. Exactly the same tools (the Larch Prover [7] and a translator [12]) are used for verifying other safety properties of design descriptions. The price for the generalization, is an increased computational complexity of the verification algorithm. However, the hierarchical verification technique compensates for this. A number of designs have been verified, including a speed-independent RAM cell, a complex switch of a data-path, and a number of standard components such as various Muller C-elements, FIFO registers, and counters.

This paper is organized as follows. Section 2 describes the design language SYNCHRONIZED TRANSITIONS used for modeling circuits. In Sect. 3 it is shown how to define and verify constraints, called invariants and protocols, on a design and its environment. Section 4 presents the definition of speed-independence, and informally describes a condition called persistency that guarantees speed-independent behavior of a design. Section 5 describes how to check the persistency condition with a combination of a theorem prover and a simple model-checking tool. Section 6 demonstrates the application of the method to the mechanical checking of a recursively described design. The appendix contains a number of definitions of concepts that are introduced informally in the main text of the paper.

2 Modeling Circuits

Speed-independence is a property of a physical circuit ensuring that the circuit operates correctly regardless of the magnitude of delays in all gates of the circuit. To make formal analysis of speed-independence possible, a model of the physical circuit is required. In this paper formal transition systems described in the design language SYNCHRONIZED TRANSITIONS are used to model physical circuits. As an example, consider a circuit component for a Muller C-element, this is described as follows:

$$\ll a = b \ \texttt{->}\ y := a \gg.$$

In this example, a, b, and y are boolean state variables, and whenever $a = b$, it is possible to assign the value of a to y. If $a \neq b$, then y keeps its current value. This construct is called a *transition*, and it models a single independent component of a circuit. A circuit with many components (operating in parallel) is described by composing a number of such transitions (one for each component).

$$\ll a \neq b \text{ -> } y := a \gg \; || \ll a := NOT\, y \gg \; || \ll b := y \gg$$

This describes a simple oscillator, if initialized in any state then the oscillator describes a computation where the three state variables a, b, and y alternate between *TRUE* and *FALSE* indefinitely. The boolean expression appearing before -> in a transition is called the precondition, when this is the constant *TRUE*, it can be omitted as exemplified by the last two transitions. State variables are introduced by a variable declaration.

$STATE\; y : BOOLEAN$

The name of the state variable (y) denotes the value of the variable. The type of the state variable (given after the ":") specifies its domain, i.e., the set of possible values. The state variable y can, for example, take the boolean values *TRUE* and *FALSE*. The value of a state variable is changed by executing a transition where the name of the state variable (e.g., y) appears on the left-hand side of an assignment (:=).

SYNCHRONIZED TRANSITIONS has a number of additional constructs that are not explained here, see [16] for a comprehensive introduction. The appendix defines the concepts used in this paper.

2.1 Operational Model

A design specifies a set of transitions (fixed throughout the computation) each of which may execute whenever it is enabled. Although a design description in SYNCHRONIZED TRANSITIONS has some similarity with a program in a high-level programming language, the interpretation is very different. An assignment statement in a high-level program is only executed when the control of the program is at the point of the statement. There is no similar global control flow determining the computations of a design in SYNCHRONIZED TRANSITIONS, which just specifies a fixed set of transitions. Unlike a Pascal or C program, the order in which transition descriptions are written does not influence the computation. SYNCHRONIZED TRANSITIONS is similar to UNITY [2] which describes a computation as a collection of conditional data-flow actions without any explicit control-flow. Operationally, the computation can be modeled as repeated non-deterministic selection and execution of an enabled transitions. In this model, transitions are executed:

- *one at a time,* i.e., only one active transition is executed in any state [1],
- *repeatedly,* each time it has been executed, it is immediately ready to be selected again,
- *independently,* of the order it appears in the design description.

[1] This corresponds to an interleaving semantics of parallel processes.

It is not required that a transition is executed immediately after it becomes enabled, because other enabled transitions may be selected. In fact, there is no upper bound on when a transition is selected. This corresponds to the unbounded gate delay in logic circuits [10]. For example, the transition $\ll y := a \ OR \ \ b \gg$ describes an OR-gate. The implicit precondition, *TRUE*, specifies that it is always allowed to set the output, y, to the logical OR of the inputs, a and b; however, an arbitrary delay may elapse between a change of the inputs and the changing of the output.

SYNCHRONIZED TRANSITIONS can be used to model circuits at different level of abstraction. The examples given above show modeling at the gate level. However, the same language is also used at higher levels, for example, a multiplier can be described as follows:

$\ll z := \ s * t \gg .$

Here s, t, and z are state variables of type integer and the transition describes a state change where the product of s and t is assigned to z.

By modeling a circuit as a design in SYNCHRONIZED TRANSITIONS, it is possible to formally verify properties of the design, for example functional properties or refinement. In this paper, the emphasis is on verifying speed-independence, and the next sections describe how to capture this property as a condition on the transition system.

3 Invariants and Protocols

To verify a certain property formally, it is necessary to formulate it rigorously in such a way that it can be determined unambiguously whether a given design has the property or not. In this paper, the focus is on formally expressing the property that a design is speed-independent, however, this is only a special case, and one can envision many other properties that a designer may wish to verify. In general, the verification tools for SYNCHRONIZED TRANSITIONS support two ways of rigorously specifying properties that are to be verified.

Invariants: are predicates over the state variables. They define a restriction on the allowable ***subset of the state space*** (the states for which the predicate holds).

Protocols: are predicates on pairs of states, ***pre, post***, defining a restriction on the allowable ***transitions between states*** (to ones where the pre- and post-state satisfy the predicate).

The language has constructs for writing invariants and protocols. As an illustration, consider an invariant stating that two state variables x and y are never *TRUE* simultaneously (mutual exclusion for x and y).

INVARIANT NOT (x AND y)

The following is an example of a protocol which states that whenever x changes, it gets the value of either y or z.

PROTOCOL $x.pre \neq x.post \Rightarrow x.post{=}y.pre$ *OR* $x.post{=}z.pre$

x.pre denotes the value of x immediately before the transition (the pre-state), and similarly *x.post* is the value of x immediately afterwards (the post-state). The same notation is used to write the value of an expression E in the pre-state as *E.pre* and in the post-state as *E.post*.

Invariants and protocols are typically used for verifying safety properties of a design, for example, mutual exclusion, or that variables in the interface follow a convention such as four-phase signaling (hence the name protocol). An in depth treatment is given in [16]. Establishing a safety property involves the following steps:

- finding a way to express a property as an invariant or/and a protocol;
- verifying that the invariant holds in any initial state of the design;
- checking that the invariant holds in all states reachable from the initial states;
- verifying that the protocol holds for any possible transition between reachable states.

Notice that invariants and protocols are stated by the designer, and that they express important safety properties that should hold for any computation of the design. It is the aim of the verification to check whether they really hold, i.e., whether they hold for any reachable state of the design and for any possible transition between reachable states. In section 4 it is is shown how speed-independence can be expressed as a protocol which may then be verified as any other protocol.

3.1 Environment and Non-determinism

In general, the computation of a design depends on the behavior of the environment. Therefore, to verify the design one needs a way of specifying the environment. In [3, 4] the behavior of the environment is expressed by the set of possible signal traces, i.e., using the same model used for specifying the internal behavior. In [1, 8] the verification is done on an autonomous circuit that is constructed by composing the original circuit with an environment making the composition autonomous. This makes it difficult to express a non-deterministic behavior of the environment. Here we describe behavior of the environment implicitly by defining protocols and invariants constraining the state space and possible transitions of *external* state variables.

By making invariants and protocols express information about permissible environments, it becomes feasible to verify a component in isolation, i.e., without checking the global behavior of the entire design. When combining several components, it is of course necessary to check that their invariants and protocols are consistent, i.e., that they have a consistent view of their interface. In [16, 17] it is shown how such a verification is done in a localized manner using the same techniques and tools that are used in this paper for checking speed-independence.

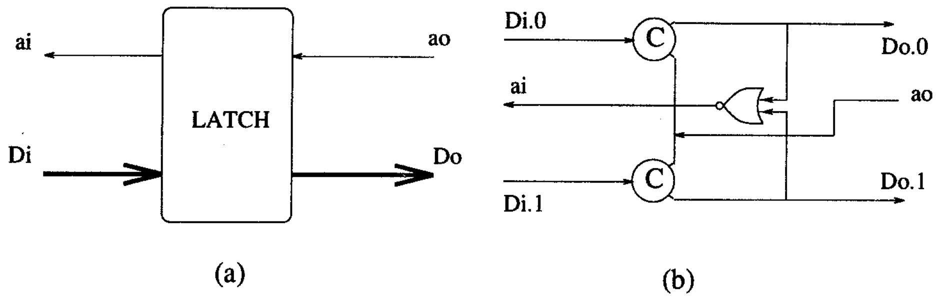

Fig. 2. The interface to the latch (a), its gate-level realization (b).

Example: A Pipeline Latch. A simple speed-independent pipeline latch is used to illustrate how to specify a non-deterministic environment. The interface to the latch consists of four state variables: two boolean acknowledgments, *ai*, *ao*, and two duals, *Di*, *Do*, modeling a one bit data-path, see Fig. 2.a. The domain for the data-path variables contains three possible values of a data bit: $\{E, T, F\}$ ("empty", "true", and "false"). The variables *ao* (the output acknowledgment) and *Di* (input data) are changed by the environment. Fig. 2.b shows a possible gate-level realization of a latch based on two C-elements and one NOR-gate [14]. The latch is described in Synchronized Transitions as follows:

```
CELL latch(ai,ao: BOOLEAN; Di,Do: dual)
BEGIN
  ≪ ai:= empty(Do) ≫ || ≪ ao ≠ empty(Di) -> Do:= Di ≫
END latch
```

The boolean function *empty* returns the value *TRUE* when the value of its dual parameter is equal to E.

The environment of the latch supplies the dual input, *Di*, and one can imagine different behaviors of the environment, it may for example, follow the return-to-empty-convention where *Di* has the value E between any two valid values. This external behavior is described by the following protocol (the predicate *same*(E) is a shorthand for $E.pre = E.post$):

$$PE_1 \equiv (NOT\ same(Di)) \Rightarrow$$
$$((ai.post \neq empty(Di.post))\ AND\ (ai.post = empty(Di.pre)))$$

This expresses that whenever *Di* changes ($NOT\ same(Di)$) then its empty status must change to the opposite value of the acknowledge *ai*. Even though this is a very restrictive protocol, it allows the environment to non-deterministically choose between the two valid values T and F.

Below a less restrictive protocol is specified where *Di* is allowed to change directly from one of the valid values to the other.

$$PE_2 \equiv (NOT\ same(Di)) \Rightarrow (ai.post \neq empty(Di.post))$$

In both cases, *ao*, must behave as follows:

$(NOT\ same(ao)) \Rightarrow (ao.post = empty(Do.post))$

To describe a pipeline latch for a wider data-path, e.g., a 16 bit word, the only change in the design description is replacing the type dual with another type. However, the gate level realization of a wider data-path would require a significant change (to compute *empty*).

4 Characterizing Speed-independence

This section gives an informal description of the persistence condition which is a protocol that must be met by a design if is going to be realized as a speed-independent circuit. In David Muller's original work, speed-independence was defined through the notion of "final classes" of behavior [13]. This paper follows the more recent trend defining a circuit to be speed-independent if its correct operation is independent of gate delays. A practically useful check for speed-independence cannot be based directly on this definition, because that would require checking a possibly infinite number of different combinations of gate delays. Instead, we have found a condition, called the ***persistency condition***, which is both mechanically checkable and sufficient to ensure speed-independence.

A transition t is persistent, if once it becomes active, it remains active, providing the same post-value for the write variable, while other transitions occur. A design meets the persistency condition, if it can be shown that all transitions are persistent both with respect to changes made by other transitions and with respect to transitions made in the environment. In the appendix it is shown how to formulate the persistency condition as a protocol which makes it possible to check that a design meets the protocol mechanically.

As an illustration, consider again the simple oscillator presented in Sect. 2.

$\ll a \neq b \ {-}{>}\ y := a \gg \ || \ \ll a := NOT\ y \gg \ || \ \ll b := y \gg$

The first transition is persistent because once it is active, it must be the case that $y \neq a$ and $y = b$. The second and third transitions are not active when this is the case, hence neither a nor b can change value. Similarly, it can be argued that the other two transitions are persistent.

To illustrate a non-persistent design consider the following modified oscillator:

$\ll y := a\ OR\ NOT\ b \gg \ || \ \ll a := NOT\ y \gg \ || \ \ll b := y \gg$

Consider a state where $a, b, y = FALSE, FALSE, FALSE$. The second transition is active, but so is the first. If the first transition changes y to $TRUE$, then the second is no longer active, and hence it is not persistent.

Example: The Pipeline Latch (continued). To illustrate the use of the persistency condition on a non-autonomous design, consider again the pipeline latch from Fig. 2. It turns out that the speed-independence of the latch depends on how strong assumptions can be made about the environment. To show that the latch design satisfies the implementation condition persistency, it is necessary to verify the persistency protocol for each of the two transitions; for brevity only the protocol for last one is shown below:

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

The left side of the implication indicates that the transition is active in the pre-state. The expression *same*(*Do*) makes the condition hold trivially for the transition itself (as stated in the appendix, this is just a way of encoding the condition $t_1 \neq t_2$). The right side of the implication requires that after the state change from *pre* to *post* the transition is still active (in the state *post*), and provides the same value for the variable *Do* (clause *same*(*Di*)).

It must be shown that for any pair of states (*pre*, *post*) satisfying the protocol, *PE*, of the pipeline latch, the persistency protocol holds. In Sect. 3.1 two different protocols are considered. For the weakest of the two protocols:

$$PE_2 \equiv (NOT\ same(Di)) \Rightarrow (ai.post \neq empty(Di.post))$$

it is *not* possible to show the persistency protocol, and hence, it cannot be expected that the pipeline latch is speed-independent when placed in an environment where only PE_2 can be assumed. Consider instead, the stronger assumption about the environment defined by PE_1:

$$PE_1 \equiv (NOT\ same(Di)) \Rightarrow$$
$$((ai.post \neq empty(Di.post))\ AND\ (ai.post = empty(Di.pre)))$$

It turns out that with this assumption, it is possible to show that the persistency protocol is met (see Sect. 5.1), and also that the other parts of the definition are met.

5 Mechanizing the Check

This section describes how to mechanize the check of the persistency condition using a combination of a theorem prover and a simple model-checker. One of the possibilities offered by these tools is the ability to check the pipeline latch separately without giving an explicit circuit realization of the environment.

The tools make it possible to take a design description like the one shown in Sect. 3.1 and automatically generate verification conditions corresponding to the persistency condition. These may then be given to the theorem prover for verification. In some cases, like the two oscillators, no additional information is needed, however, in more interesting cases, it is necessary to provide an invariant excluding some or all of the states that the design will never enter. For the latch, it is for example the case that:

$$(NOT\ empty(Do))\ AND\ (NOT\ empty(Di)) \Rightarrow Di{=}Do$$

This information may be added as an explicit invariant, and for the simple latch example it is possible to manually generate the few extra invariants needed (part of it is shown above), but in larger and more complicated examples, it can be a significant help to use a model-checking tool to characterize the reachable part of the state space; this is explained further in the next section.

5.1 Reachability Invariant

The *reachable state space* of a design is usually a small subset of the state space defined by the cartesian product of the domains of all state variables. A characterization of this reachable state space is useful for almost any kind of formal verification, and also for verifying speed-independence which is the topic of this paper. One way to characterize the reachable state space is as an invariant, called the *reachability invariant*. Below, it is discussed how to generate this invariant.

Let $\mathcal{D}$ be a design in SYNCHRONIZED TRANSITIONS with the set of transitions $t_1, ..., t_n$, the external invariant I_E, and the external protocol P_E. Each transition, t_i, defines a state transition predicate $t_i(pre, post)$. In other words, each transition t_i defines a state transition relation R^{t_i}. The union of these relations define a state transition relation for the *internal* state variables of the design. This relation, R, is defined as follows:

$$(S_1, S_2) \in R \Leftrightarrow I_E(S_1) \wedge I_E(S_2) \wedge (P_E(S_1, S_2) \vee \exists t_i \; t_i(S_1, S_2))$$

Two states S_1 and S_2 are in relation R if they differ either by the value of an external variable or by the value of the write variable of transition t_i. This variable can change its value from state S_1 to state S_2 either as defined by $P_E(S_1, S_2)$ or by $t_i(S_1, S_2)$.

The transitive closure of R, denoted by R^*, is called the *reachability relation* of the design. If the initial predicate, U_0, is given, then the set of states reachable from the initial states can be characterized by a predicate, called the *reachability invariant*, I_{U_0}, that is defined as follows:

$$I_{U_0}(S) \Leftrightarrow \exists u_0 \in U_0 \; (u_0, S) \in R^*$$

For simple designs, derivation of this invariant can be done manually, but for more challenging designs this is too laborious. However, the reachability invariant can be derived automatically using a model-checking tool. For our experiments we have used the state generation kernel of the TRANAL system, included in the FORCAGE system [8]. The algorithm is based on a breadth-first search of the state transition graph, this is illustrated in Fig. 3. The states of a design are represented as boolean vectors and transitions as boolean vector representations of boolean functions. The algorithm proceeds in stages. At the kth stage a kth layer of states is derived. All states in this layer are reachable from the initial set of states through k transitions. For the next iteration, the states reachable from kth layer are found, thus giving the $(k+1)$-th layer of states that are reachable in $k+1$ transitions. The algorithm ends when no more new states can be found. To simplify the check for the fixed point of the generation process, a heuristic is described in [8] allowing one to compare two layers of the reachability set instead of performing comparison of the whole set of states in the current layer.

State explosion is a potential danger, of model-checking, however when used as here to generate invariants of a modular design, we hope to avoid the problem. The individual cells of a modular design contain relatively few state variables and model-checking is only used on such limited modules, whereas the composition of modules is verified using localized verification. However, our experience with the approach is still limited.

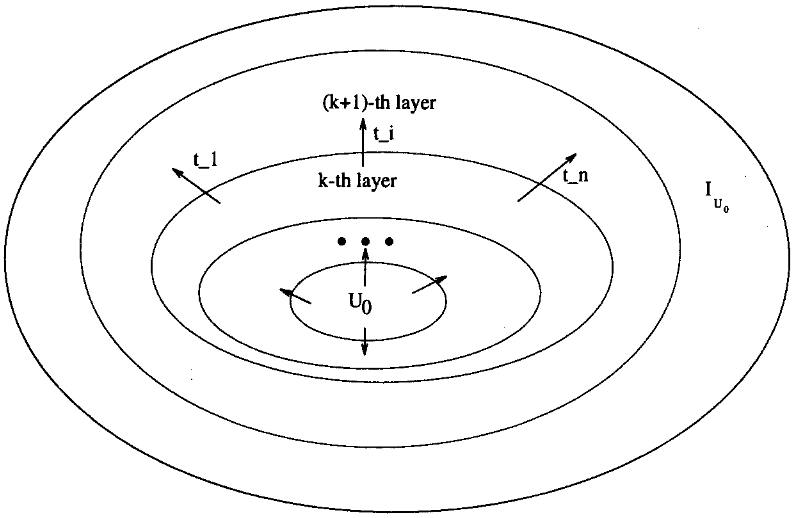

Fig. 3. **Breadth-first search for reachability invariant**

Example: The Pipeline Latch (continued). The following expression characterizes the invariant derived automatically for the pipeline latch:

INVARIANT
(Di = Do) OR (ai AND empty(Do)) OR ((NOT ai) AND empty(Di))

This invariant, called I below, can be used to verify that the pipeline latch meets the persistency condition. If the invariant is inserted in the design description for the pipeline latch and if this is given to the automatic translator, the verification conditions shown below are generated (they have been simplified a little to make this presentation more clear). The restrictions on the interface constrains the environment, but it must also be met by the latch itself, in Sect. 4, this restriction was defined by the following protocol (PE).

PROTOCOL
$((NOT\ same(ao)) \Rightarrow (ao.POST = empty(Do.POST))) \wedge$
$((NOT\ same(Di)) \Rightarrow (ai.POST \neq empty(Di.POST)) \wedge$
$(ai.POST = empty(Di.PRE)))$

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 protocol *PE* describing the environment.

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

Where $Persistent^{t_j}(pre, post)$ is a protocol describing the persistency condition for transition t_j, $i, j \in 1, 2$. **End of example**

The verification technique described above has been used to check a number of designs, for example, a complicated switch of a data-path, various realizations of

```
(* latch *)
  ≪ am:= NOT (p  OR  q) ≫ ||
  ≪ a = pm0  -> p:= a ≫ ||
  ≪ a = qm1  -> q:= a ≫ ||
(* OR-gate *) ≪ pm0:= pm  OR  qm0 ≫ ||
toggle(qm, qm0, qm1) ||
{ N = 1 | count_one(am, pm, qm) } ||
{ N > 1 | count_even(am, pm, qm, N DIV 2) }
```

Fig. 4. The modulo-2^n counter, $N = 2^n$

a Muller-C element, and a speed-independent RAM design. In the next section, it is shown how a parameterized recursive design description can be checked for speed-independence without unfolding the recursion or binding the parameters to particular values.

6 Verification of Modular Designs

In this section it is shown how localized verification [17] is used to check a modular, parameterized, and recursively defined design. The modulo-N counter is an example of such a design; the value of N is left unspecified in the description which is a recursive composition of a modulo-$N/2$ counter, a toggle, and a few simple gates. A 2^n counter can be viewed as a composition of a 2^{n-1} counter and a counter cell, as shown in Fig. 1. The counter cell is composed of a toggle element, a pipeline latch (from Fig. 2.b), and an OR-gate. A SYNCHRONIZED TRANSITIONS description of a modulo-2^n counter is given in Fig. 4. The complete counter that works for arbitrary numbers has been designed and verified, it is omitted in this paper, because it does not show anything substantially new compared to the modulo-2^n counter.

The design shown in Fig. 4 is the same as the one shown in Fig. 1. The first three transitions correspond to the latch (shown in Fig. 2) and the fourth transition is the OR-gate. The last three lines describe the sub-cells. The first is a toggle element. Then follows two conditional instantiations, if $N > 1$ a new instance of the count-even cell is made (the modulo-$N/2$ counter) and if $N = 1$ the recursion stops by instantiating a simple modulo-1 counter.

Using the approach described in this paper, the verification of the speed-independence of the modulo-2^n consists of the following steps:

1. describe the cell interfaces by protocols (and invariants),
2. generate the persistency protocol for each cell,
3. generate the reachability (or another sufficiently strong) invariant of each cell,
4. generate the verification conditions,
5. verify the verification conditions.

The last four steps can be done mechanically whereas the first step requires some insight in the design in order to capture all essential aspects of the environment of a cell. Each step is now illustrated using the modulo-2^n counter as an example.

Protocols for the Cell Interfaces. When the interface variables of the counter cell change, it is always known what the new value is, e.g., whenever p (or q) change it becomes equal to a. This is described by the following protocol:

PROTOCOL
  NOT same(p) $\Rightarrow$ *p.post* $=$ *a.pre AND*
  NOT same(q) $\Rightarrow$ *q.post* $=$ *a.pre AND*
  NOT same(a) $\Rightarrow$ *a.post* $\neq$ *(p.pre OR q.pre)*

Similar protocols are needed to describe the interfaces to the toggle cell and the modulo-1 counter. It requires some experience and insight in the design to come up with these protocols. If they are too weak (does not contain enough information) it will not be possible to verify the desired properties and the entire verification cycle must be repeated.

The Persistency Protocol. It can be verified that the counter is speed-independent by showing that it meets definition 4. This is done by including persistency protocols in the design description, for example, in the cell *count_even*, the persistency protocol for the four transitions is:

((am.pre≠NOT(p.pre OR q.pre) AND SAME(am)) $\Rightarrow$
    (am.post≠NOT(p.post OR q.post))) AND
(((a.pre=pm0.pre) AND (p.pre≠a.pre) AND SAME(p)) $\Rightarrow$
    ((a.post=pm0.post) AND (p.post≠a.post))) AND
(((a.pre=qm1.pre) AND (q.pre≠a.pre) AND SAME(q)) $\Rightarrow$
    ((a.post=qm1.post) AND (q.post≠a.post))) AND
((pm0.pre≠(pm.pre OR qm0.pre) AND SAME(pm0)) $\Rightarrow$
    (pm0.post≠(pm.post OR qm0.post)))

Similar protocols are needed in the other cells (toggle and count-one).

The Reachability Invariant. To verify that the design meets the consistency protocols (and the other protocols formulated by the designer) it is necessary to find an invariant for each cell. The toggle and the count-one cells are simple enough that a suitable invariant can be formulated by the designer. However, for the count-even cell, it is possible to generate the following invariant automatically as described in section 5.1:

```
INVARIANT
  ((NOT pm) AND (NOT qm0) AND (NOT p) AND (NOT pm0) AND
      ((am AND qm AND qm1) OR
  ((NOT am) AND ((NOT qm) AND (NOT qm1) OR (q AND qm1)))))
  OR ((NOT q) AND (NOT qm1) AND (am AND (NOT pm) AND qm AND
      ((pm0 AND qm0) OR
  ((NOT p) AND (NOT pm0)))
  OR (pm0 AND qm0 AND p AND (NOT am) AND (NOT pm)) OR
  ((NOT qm) AND (NOT qm0) AND (pm AND pm0 AND (am OR p) OR
  (am AND (NOT p) AND (NOT pm0)) OR
(p AND (NOT am) AND (NOT pm))))))
```

This is by construction an invariant, and hence, it need not be verified again, it is however included in the design description and utilized to verify the persistency and other protocols.

The Verification Conditions. To verify that the modulo-N counter meets the persistency protocols the design description is translated into a number of verification conditions for the theorem prover. The localized verification technique embedded in the translator utilizes the modular structure of the design; therefore, the number of verification conditions that are generated to verify speed-independence (or any other safety property that can be described as an invariant or protocol) is linear in the size of the design description. In case of the modulo-N counter, the number of verification conditions is independent of N (which determines the depth of recursion).

First, all transitions of each cell are verified locally by showing that the (local) invariants and protocols of that cell (including persistency) will hold after executing the transition, assuming that the invariant held in the pre-state. This is a standard way of verifying invariance [6].

The second step is to verify that no cell instantiation results in a design where the invariants and protocols for the instantiated cell are violated by the environment and vice versa. This is done using the localized verification technique [17] without considering the individual transitions of the cell or the environment. Instead the invariants and protocol of the instantiating cells are assumed to express sufficient constraints on the state changes of the environment to show an implication ensuring that the protocols and invariants of the instantiated cell hold. For example, to verify the instantiation of the cell *toggle* the following two implications must be shown:

$$I_{ce}(pre) \wedge I_t(pre) \wedge I_t(post) \wedge P_t(pre, post) \Rightarrow I_{ce}(post) \wedge P_{ce}(pre, post)$$

$$I_{ce}(pre) \wedge I_t(pre) \wedge I_{ce}(post) \wedge P_{ce}(pre, post) \Rightarrow I_t(post) \wedge P_t(pre, post)$$

I and P are the invariants and protocols in the toggle cell (subscript t) and the count-even cell (subscript ce). Note that the two conditions avoid considering the individual transitions of the two cells. Two similar conditions are generated for each cell instantiation in the design. Note also that the protocol, for example

P_t, includes the persistency protocols. Hence, these proof obligations cover part 2 of definition 4.

A more detailed description of the localized verification technique is given in [17] which also contains a proof of the soundness of the technique.

The Verification. The verification of the counter with the mechanical tools shows that the persistency condition is met; therefore, the counter design is speed-independent. The complete verification of the counter with the LARCH PROVER on a DEC Alpha station took approximately 22 min, and the run time is independent of the actual size of the counter (parameter N).

A drawback of theorem provers is that they often require intensive human interaction during the verification. Our translator generates a script that controls the verification and usually only a little additional manual interaction is needed. To verify speed-independence of the modulo-N counter, it was necessary to manually help the theorem prover approximately once, for each verification condition.

7 Conclusion

This paper has described a set of mechanical tools that can be used to assist a designer in verifying that a design is speed-independent. There are already model-checking tools available for this. However, as demonstrated above, the use of general purpose tools gives some new possibilities like checking high-level and incomplete designs. Furthermore, the use of general purpose theorem provers avoids the need for constructing specialized tools. Although the run-time for the method described in this paper is worse than for checking speed-independence by known model-checking tools, we find that it is still reasonable for practical applications. For example, all components of a vector multiplier design were verified within a few hours of run-time on a DEC Alpha station. Several mistakes in the design were found. The inherent universality of theorem provers allows one to verify any properties expressible in the logic theory underlying the particular theorem prover. The persistency condition is an example of a safety property, and any other safety property can be mechanically verified using the same combination of a theorem prover and model-checker.

Appendix. Notation and Terminology

A design consists of a set of state variables, $s_1, s_2, \ldots, s_n$, and a number of transitions, $t_1, t_2, \ldots, t_m$. For a given design, the set of state variables and transitions are fixed. In this paper, we focus on a subset of SYNCHRONIZED TRANSITIONS in which all variables are of finite types, i.e., each variable has a finite set of possible values from a fixed finite domain (determined by the type).

Each transitions, t, has a form $\ll C_t$ -> $z_t := E_t \gg$, 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. The transition t is *enabled* (in a state s) if and only if C_t is satisfied (in s). The transition t is *active* (in a state s) if and only if it is enabled and $z_t \neq E_t$ (in s). For each transition, t, the predicate $active^t$ is defined as follows:

$$active^t \equiv C_t \wedge (z_t \neq E_t)$$

When necessary, this predicate is explicitly applied in a particular state, s, and then 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. More formally, each transition t defines a predicate $t(pre, post)$ on the pairs of states such that: $active^t(pre)$ and state $post$ differs from the state pre only by the value of the variable z_t, i.e., $z_t.post = E_t.pre$, where $z_t.post$ 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. The predefined predicate $same(x)$ is a shorthand for $x.pre = x.post$, where x can be any state expression.

A design defines a set of computations that are sequences of states, $S_0, S_1, \ldots$, where S_0 is an initial state, and for each pair, S_i, S_{i+1}, there is a transition, t, such that $t(S_i, S_{i+1})$, i.e., S_i is a pre-state of t, and S_{i+1} is a post-state of t. Note, that there is only a single transition accounting for one step of the computation (going from S_i to S_{i+1}). Alternatively it might be external variables of the design that change value between S_i and S_{i+1} according to the external protocol.

The *initial state* (or set of states) is specified by defining the initial value of some or all of the state variables.

INITIALLY $a = FALSE\ b = FALSE$

State variables get their initial value before any transitions are executed, and they retain this value until a different value is assigned to them. It is not required that all state variables are given an initial value.

A1. Well-behaved Designs

To ensure a one to one correspondence between a design description in SYNCHRONIZED TRANSITIONS and its circuit realization, it is necessary to exclude certain designs, for example, those that contain contradictory assignments to the same state variables.

The *write set*, W^t, of a transition, t, is the set of state variables appearing on the left-hand side of assignments. In this paper multi-assignments are not considered, therefore the write set has a single element: $W^t = \{z_t\}$. 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.

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 the 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 SYNCHRONIZED TRANSITIONS design is called *well-behaved* if it obeys the exclusive write and unique write conditions.

It can be shown that the expressiveness of well-behaved SYNCHRONIZED TRANSITIONS designs is enough to code any asynchronous logic circuit.

A2. Characterizing Speed-independence

Let t be a transition of a design. The protocol $Persistent^t(pre, post)$ is defined as follows:

$$Persistent^t(pre, post) \equiv Active^t(pre) \Rightarrow (Active^t(post) \wedge same(E_t))$$

Intuitively, $Persistent^t$ defines the constraint that transition t stays active, providing the same post-value for the write variable, while other transitions occur. If the write variable of the transition t is of type boolean, then the latter conjunct, $same(E_t)$, is redundant.

Definition 4. Let D be a design with the invariant I_U and the protocol $P_E(pre, post)$. Then D satisfies the *persistency* condition, if the following can be shown:

1. for all pairs of transitions t_1, t_2 in D, $t_1 \neq t_2$:
   $t_1(pre, post) \wedge I_U(pre) \Rightarrow Persistent^{t_2}(pre, post)$
2. for any transition t in D:
   $P_E(pre, post) \wedge I_U(pre) \Rightarrow Persistent^t(pre, post)$.

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. The justification of the persistency condition is given in [9], where it is argued that *a well-behaved* SYNCHRONIZED TRANSITIONS *design is speed-independent, if and only if it satisfies the persistency condition.*

The persistency protocol generalizes the notion of a *conflict state* [8], that is used for analysis of semi-modular binary circuits.

In the mechanical tools described in this paper the following formulation of the protocol $Persistent^t(pre, post)$ is used:

$$\begin{aligned}&Persistent^t(pre, post) \equiv \\ &\quad Active^t(pre) \wedge same(W^t) \Rightarrow (Active^t(post) \wedge same(E_t))\end{aligned}$$

An additional clause $same(W^t)$ guarantees that transition t has not occurred between states *pre* and *post*. This clause ensures that the $t_1 \neq t_2$ condition holds as required by definition 4.

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. K. Mani Chandy and Jajadev Misra. *Parallel Program Design: A Foundation.* Addison-Wesley, 1988.
3. D.L. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits.* The MIT Press, Cambridge, Mass., 1988. An ACM Distinguished Dissertation 1988.
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. Jo C. Ebergen and Ad M. G. Peeters. Design and analysis of delay-insensitive modulo-N counters. *Formal Methods in System Design*, 3(3), December 1993.
6. R.W. Floyd. Assigning meanings to programs. In J.T. Schwartz, editor, *Proceedings of the Symposium in Applied Mathematics*, volume 19, pages 19–32. American Mathematical Society, 1967.
7. John V. Guttag, James J. Horning with S.J. Garland, K.D. Jones, A. Modet, and J.M. Wing. *Larch: Languages and Tools for Formal Specification.* Springer-Verlag Texts and Monographs in Computer Science, 1993. ISBN 0-387-94006-5, ISBN 3-540-94006-5.
8. M. A. Kishinevsky, A. Y. Kondratyev, A. R. Taubin, and V. I. Varshavsky. *Concurrent Hardware. The Theory and Practice of Self-Timed Design.* John Wiley and Sons Ltd., 1994.
9. Michael Kishinevsky and Jørgen Staunstrup. Checking speed-independence of high-level designs. In *Proceedings of the Symposium on Advanced Reserch in Asynchronous Cirsuits and Systems*, Utah, USA, November 1994. to appear.
10. L. Lavagno and A. Sangiovanni-Vincentelli. *Algorithms for synthesis and testing of asynchronous circuits.* Kluwer Academic Publishers, 1993.
11. Alain J. Martin, Steven M. Burns, T. K. Lee, Drazen Borkovic, and Pieter J. Hazewindus. The first asynchronous microprocessor: the test results. *Computer Architecture News*, 17(4):95–110, June 1989.
12. Niels Mellergaard. *Mechanized Design Verification.* PhD thesis, Department of Computer Science, Technical University of Denmark, 1994.

13. D. E. Muller and W. C. Bartky. A theory of asynchronous circuits. In *Annals of Computing Laboratory of Harvard University*, pages 204–243, 1959.
14. David E. Muller. Asynchronous logics and application to information processing. In H. Aiken and W. F. Main, editors, *Proc. Symp. on Application of Switching Theory in Space Technology*, pages 289–297. Stanford University Press, 1963.
15. Christian D. Nielsen. *Performance Aspects of Delay-Insensitive Design.* PhD thesis, Technical University of Denmark, 1994.
16. Jørgen Staunstrup. *A Formal Approach to Hardware Design.* Kluwer Academic Publishers, 1994.
17. Jørgen Staunstrup and Niels Mellergaard. Localized verification of modular designs. *Formal Methods in System Design*, 1994. accepted for publication.
18. Kees van Berkel, Ronan Burgess, Joep Kessels, Ad Peeters, Marly Roncken, and Frits 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.