SCS&E Report 9305
April, 1993

Signal Transition Graph Constraints for Synthesis of Hazard-Free Asynchronous Circuits with Unbounded-Gate Delays

Radhakrishna Nagalla and Graham Hellestrand

SCHOOL OF COMPUTER SCIENCE AND ENGINEERING
THE UNIVERSITY OF NEW SOUTH WALES

Abstract

A synthesis procedure for asynchronous control circuits from a high level specification, signal transition graph (STG), is described. In this paper, we propose some syntactic constraints on STG to guarantee hazard-free implementation. We have introduced a global persistency concept in order to establish the relationship between the persistency concept introduced by Chu [2] (which we call local persistency) and the consistent state coding (CSC). The STG syntactic constraints required to compute the *input set* of a signal are identified. We analyze all hazards under both single and multiple input change conditions and propose necessary changes to the net contraction and logic synthesis procedures. The proposed changes are guaranteed to generate hazard-free circuits with the unbounded-gate delay model, if the STG is live, safe and has consistent state coding.

1 Introduction

Synthesis of asynchronous circuits is gaining popularity due to inherent clock distribution and clock skew problems associated with large synchronous circuits. The asynchronous design approach eliminates the need for global clock synchronization and circumvents problems due to clock skew. In synchronous systems, the clock cycle accommodates the slowest combinational path and all operations start with a global clock. In asynchronous systems, an operation starts whenever certain operations (events) are completed. Thus in asynchronous systems, the system latency is decided by the average delay rather than the worst case delay as with synchronous systems.

Despite many advantages of asynchronous circuits, they have not been extensively used because of the lack of good specification and synthesis tools. The synthesis of asynchronous circuits is difficult because of the presence of the hazards (transient errors in a circuit due to the presence of stray delays). Asynchronous logic has been designed traditionally using finite state–machine like descriptions called flow tables [12, 16]. Hazards were removed by introducing redundant states during the state assignment and/or by adjusting the feedback delays. However, the traditional state based approach is impractical in designing large scale asynchronous circuits due to the exponential state table size and the difficulty in adjusting the relative delays of the gates.

Recently Chu [2, 3] introduced a formal, technology independent, graphical specifications called signal transition graphs (STGs) which specify the circuit behaviour using the causal relations between the signal transitions rather than the states. An STG is based on a type of Petri net called the free–choice net which can explicitly describe the major aspects of asynchronous control circuit behaviour such as concurrency, causality and conflict.

Chu developed a promising synthesis technique to implement asynchronous control circuits from formal STG specifications. He proved that the constraints on STG specifications such as liveness, persistency and consistent state assignment are necessary and sufficient to implement hazard–free asynchronous control circuits. Meng et al. [10, 11] developed algorithms for automated synthesis of asynchronous circuits, in which another notion of the persistency called semi–modularity was taken into account. Vanbekbergen et al. [17, 18] developed methods for the optimized synthesis of STG specifications by considering state coding properties. Luciano et al. [7, 8, 9] and Moon et al. [13] analyzed the possible hazards in asynchronous circuits and developed methods for their removal.

In [7], Luciano proved, contrary to Chu's claim, that the persistency is not a necessary condition and the complete state coding (CSC) is the only necessary and sufficient condition for hazard–free circuit implementation. Recently Puri [15] presented proofs establishing a relationship among Chu's persistency constraint, the complete state coding and the signals having a *controlling value* (definition in Section 2.4) in another signal. The fact that a signal having a controlling value in another signal can be determined only after realizing the logic.

In this paper, we are concerned with the STG constraints for the hazard–free synthesis of asynchronous circuits with a unbounded gate–delay model (wires have no delay and gates have unbounded delay). We show that in some cases, the definition of the persistency given by Chu does not represent persistency (literal meaning) of the signals in the STGs. It can be easily shown that a CSC violation is always associated with a non–persistent (literal sense) signal. We have provided an extension to Chu's persistency constraint, called global persistency constraint in order to always ensure the persistency (literal sense) of the signals in the STG. The global persistency has one to one correspondence to the CSC property of the state graphs.

We also provide the STG syntactic constraints necessary to detect those signals having controlling value in another signal. We prove that the net contraction [3] does not produce any solution more efficient than the solutions obtained using existing boolean minimisation techniques. Finally, we give a thorough analysis of the hazards under multiple input change (MIC) conditions and prove that hazard–free circuit implementations (under both single input change and multiple input change conditions) can always be obtained.

This paper is organized as follows. Section 2 presents some basic definitions from the literature and describes some previous work in the area. Section 3 deals with STG syntactic constraints such as liveness, persistency and consistent state coding (CSC). Section 4 establishes the relationship among Chu's persistency constraint, global persistency constraint and the CSC. Section 5 discusses the relationship between the controlling value and the CSC. Section 6 gives an analysis on the net contraction. Section 7 analyzes hazards under both single and multiple input change conditions and provides appropriate changes to the net contraction and logic synthesis procedures in order to remove all the hazards. Section 8 describes two examples, applying the ideas presented in the paper. Section 9 concludes the paper.

2 Preliminaries

2.1 Signal Transition Graph (STG)

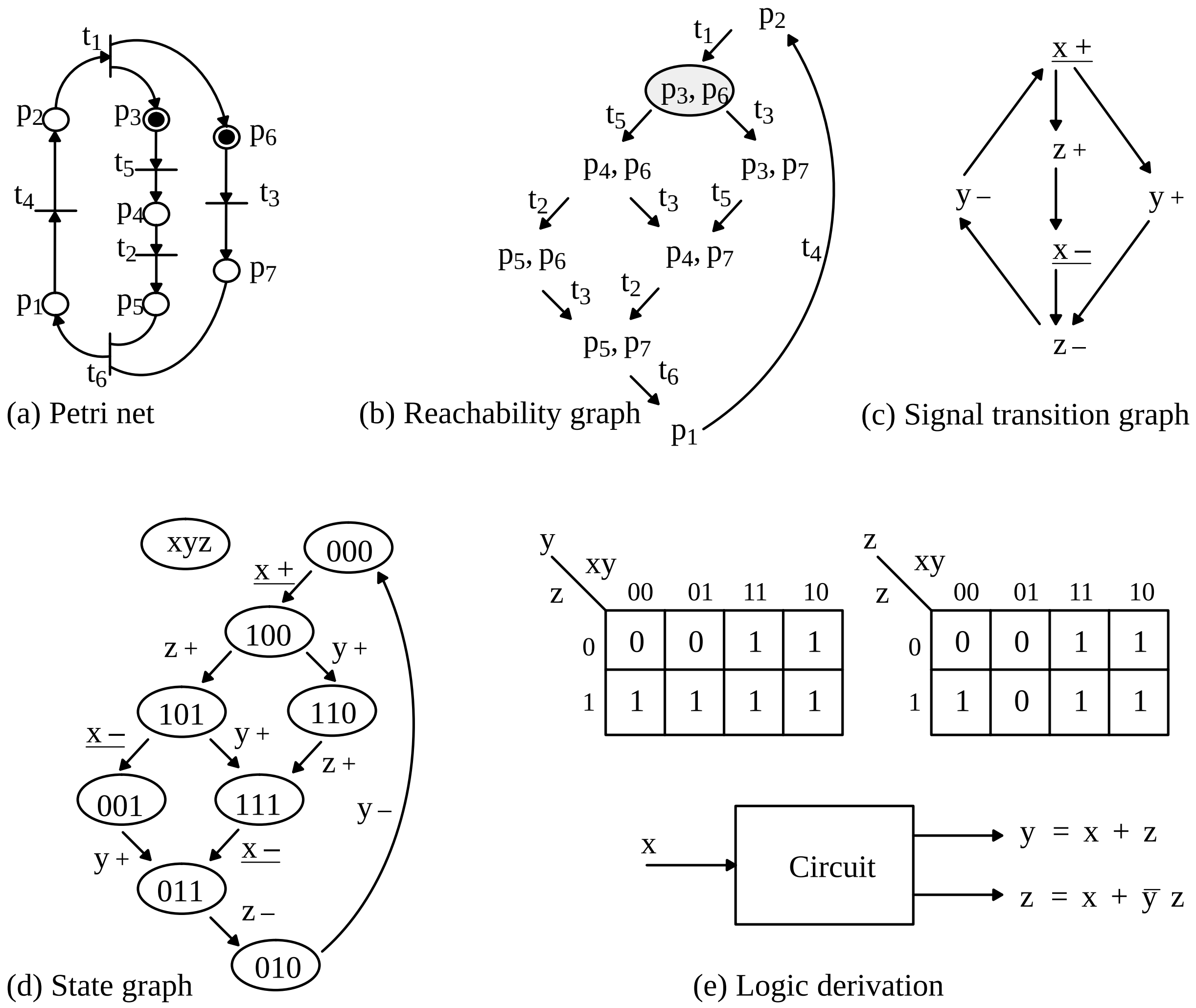
An *STG* is an *interpreted* free–choice Petri net introduced by Chu [3] for specifying self–timed asynchronous control circuits. STGs can be used effectively to specify the asynchronous interface circuits, because the causal relations between the signal transitions can easily describe explicit concurrency, sequencing and conflict behaviour.

2.1.1 Petri Nets

A *Petri net* is a four–tuple $< T, P, F, M_o >$ where T, P and F form a directed bipartite graph and M_o is called the initial *marking*. In the graph, P is a set of *places* which can be used to specify conflict or choice, T is a set of net *transitions* and the directed edges F give the *flow relation* between transitions and places; i.e $F \subseteq (T \times P) \cup (P \times T)$. A marking M is a collection of places corresponding to the local conditions which hold at a particular moment. A marking is graphically represented by solid circles called tokens residing in these places and is an assignment of nonnegative integers to denote the number of tokens residing at each place). An example Petri net is shown in Figure 1(a) in which transitions are drawn as bars, places as circles, and the flow relation as directed arcs. Transitions are usually interpreted as *events* in a control system while places are interpreted as the *local conditions* which become true or cease to be true due to the occurrence of some events, as specified by the flow relation.

A transition t is called an *output* transition of a place p if $(p, t) \in F$. Similarly a transition t is called *input* transition of a place p if $(t, p) \in F$. Likewise, *input* places and *output* places are also defined. For a place p, $.p$ and $p.$ are often referred to as sets of input and output transitions. Similarly, for a transition x, $.x$ and $x.$ are often referred to as sets of input and output places respectively.

A Petri net structure describes the static behaviour of a control system. Its dynamic nature is obtained by its markings and by firing its transitions which transforms one marking to another. A transition is said to be enabled if all its input places have at least one token. An enabled transition must eventually fire and its firing removes one token from each input place and adds one token to each output place. The result of the execution of the net is described by a form of finite automata called *reachability graphs* as shown in Figure 1(b). In a reachability graph, each node represents a state corresponding to a marking of the net, a labeled arc between nodes indicates the transition from one marking to another due to the firing of an enabled transition. The state corresponding to the initial marking of the net is highlighted in Fig. 1(b). A good review of more basic Petri net concepts is given in [14].



(a) Petri net

(b) Reachability graph

(c) Signal transition graph

y: z \ xy	00	01	11	10
0	0	0	1	1
1	1	1	1	1

z: z \ xy	00	01	11	10
0	0	0	1	1
1	1	0	1	1

$y = x + z$

$z = x + \overline{y}\, z$

(d) State graph

(e) Logic derivation

Figure 1. An example illustrating the synthesis of a circuit from a Petri net specification

2.1.2 Free–Choice Nets

A free–choice net (FC net) is a Petri net where the input place *p* must be the unique predecessor of $t_{1,}t_2 \ldots t_n$, if more than one transition $t_1 \ldots t_n$ share the same *p*. Such a *p* is called a *free–choice* place. A marked graph (MG) is a net in which each place has at most one input transition and at most one output transition. A state machine (SM), which is a dual notion of a marked graph, is a net in which each transition has at most one input place and at most one output place. Marked graphs represent the structure of concurrent systems with deterministic choices whereas state machines represent the structure of sequential systems with non–deterministic choices [3,6].

2.1.3 Live and Safe Free–Choice Nets

An FC net is live if every transition can be enabled through some sequence of firings from the initial marking M_o. It can be shown that in a live FC net, each transition can be enabled infinitely often [4]. It can also be shown that an FC net is live if and only if the token count of every simple loop of the net is nonzero [4]. Liveness guarantees that no deadlock occurs in the circuit.

An FC net is safe if every place contains no more than one token for all markings reachable from the initial marking M_o. This means that in a safe FC net, a transition cannot fire twice in a row without firing some other transition. An FC net is safe if and only if every arc in the net is in a simple loop with a token count of one [4].

Hack [6] proved that a live and safe FC net can be decomposed into either a set of SM components or a set of MG components such that each transition and place can be represented in at least one SM component or MG component.

2.1.4 STGs as Interpreted Free–Choice Nets

An STG is an interpreted free–choice petri net where the transitions are interpreted as value changes on input, output or internal signals of the specified circuit. The transitions are described by $t \times \{+,-\}$ where *t*+ represents a 0 to 1 transition and *t*– represents a 1 to 0 transition. From now on, *t** will denote any transition (i.e either *t*+ or *t*–) and *t**~ will denote its complementary value (i.e either *t*– or *t*+). In an STG, transitions are represented by their names instead of a bar and a label. Figure 1(c) shows an STG of a control circuit [10] whose underlying FC Petri net is given in Figure 1(a). The FC net transitions $t_1, t_2, t_3, t_4, t_5, t_6$ represent the STG signal transitions *x*+, *x*–, *y*+, *y*–, *z*+, *z*– respectively.

In STGs, the input signal transitions are underlined to distinguish from the internal and the output signal transitions. The reason is, input signals are generated by the *environment* where as the output and internal signals are generated by the *circuit to be synthesized*. Each place with a single input and output transition is replaced by a directed arc representing the causal relationship between the transitions. For example, the place p_5 between the transitions t_2 and t_6 in Figure 1(a) is represented as $x- \longrightarrow z-$ in Figure 1(c), meaning that the transition *z*– always follows the transition *x*–.

Two transitions t_1 and t_2 in an STG are :

- *ordered*, if they are never enabled in the same marking. i.e. if there exists a simple loop containing both of them. For example in Figure 1(c), transitions *x*– and *z*+ are ordered. If *t1* and *t2* are ordered, they can be represented either as $t1 \rightarrow t2$ or *as* $t1 \xrightarrow{*} t2$. An arc $t1 \rightarrow t2$ constraints the signal transition *t2* to be an immediate successor of *t1*. $t1 \xrightarrow{*} t2$ means *t2* is a successor of *t1*.
- in *conflict*, if firing one transition disables the other transition and vice–versa.
- *concurrent*, if there exists a reachable marking where both of them are enabled and none of the two transitions are disabled by the firing of the other transition. i.e. if there is no simple loop containing both of them and if they are not in conflict. For example in Figure 1(c), transitions *y*+ and *x*– are concurrent.

AN STG is defined as *live* [3], if and only if :

- the underlying net is live and safe.
- for each signal *t* there is at least one SM component, initially marked with one token, such that it contains all transitions *t** of *t*, and each path from a transition *t** to another transition *t** (i.e. both raising or falling) contains a complementary transition *t**~. This condition ensures that the interpreted finite automata (i.e. the state graph) has a consistent state assignment.

2.2 State Graph

A state graph (SG) is an equivalent finite automata obtained from all the possible transition sequences defined by a signal transition graph. The SG is a directed graph, where each node called a state, has a one to one correspondence with a live and safe marking of the STG. It is defined formally as a 2–tuple $< V, E >$, where V is a set of states and E is set of edges. i.e.

$E \subseteq (V \times V)$. The SG can be derived from a live STG using a deterministic procedure given in [3].

In order to implement the SG, the states are interpreted as binary vectors representing values of the signals in a circuit. Figure 1(d) shows the state graph of the STG given in Figure 1(c). The control circuit is comprised of the set of signals $J = \{x, y, z\}$. Hence in its state graph, every state is represented as binary vector $< x, y, z >$. In the SG, the concurrent transitions of the STG are explicitly represented as all the possible transition sequences.

2.2.1 Consistent State Coding

The synthesis procedure described in [3] uses the signals in a circuit directly as state variables, so that the circuit must be able to tell its global state only from its input and output signals. When two different states are given the same binary representation, the digital circuit cannot distinguish the two states from each other. Thus every state of the SG must be assigned a unique binary vector of the signal values. A state graph is said to satisfy *consistent state assignment* (proposed by Chu in [3]) if the binary vector V_i of every state in the SG meets the following condition

> An edge with a transition $t+(t–)$ from state s_1 to state s_2 (i.e. $s_1 \xrightarrow{t+(t-)} s_2$) defines the value of signal t equal to 0(1) in V_1 and 1(0) in V_2 respectively. If the edge $s_1 \rightarrow s_2$ does not have a transition t^*, then the signal t must have the same value in both V_1 and V_2. The value of signal t in V_2 is called the *implied value* of t in V_1.

If no two different states are assigned the same binary code in a SG, then the SG is said to possess *unique state coding* (USC) property [17].

The *complete state coding* (CSC) [13] property is same as the *consistent state assignment* proposed by Chu in [3] and is formally defined as follows.

A state graph is said to satisfy *complete state coding* constraint if :

- no two states are assigned the same binary code
- the transitions of non–input signals, enabled in two states with the same binary state assignment, are same.

Thus, only the input transitions enabled in two states with same binary code are different, and it is assumed that the environment can distinguish between them. In order to implement a circuit, the state graph must satisfy CSC property [13]. The state graph in Figure 1(d) satisfies the CSC property and thus the logic functions for the output signals *y* and *z* are derived as shown in Figure 1(e).

2.3 Hazards

A *hazard* is a possible deviation of the output from the expected behaviour with respect to some input change. Hazards occur due to stray delays in the circuit. Combinational hazards can be classified into two categories namely static and dynamic hazards.

A *static* hazard is a $0 \rightarrow 1 \rightarrow 0$ or a $1 \rightarrow 0 \rightarrow 1$ transition in any condition where no such a transition for the signal is specified. The former type of hazard is called a static 0–hazard and the latter a static 1–hazard.

Dynamic hazards occur when the expected behaviour is a single transition $0 \rightarrow 1$ or $1 \rightarrow 0$ but the possible transition becomes $0 \rightarrow 1 \rightarrow 0 \rightarrow 1$, $0 \rightarrow 1 \rightarrow 0 \rightarrow 1 \rightarrow 0 \rightarrow 1$, etc. or $1 \rightarrow 0 \rightarrow 1 \rightarrow 0$, $1 \rightarrow 0 \rightarrow 1 \rightarrow 0 \rightarrow 1 \rightarrow 0$, etc.

Hazards are associated with circuit configurations, not with physical circuits. Thus the circuits should be implemented in such a way that all static and dynamic hazards are eliminated.

2.4 Logic functions

A *single–output logic function f* of *n* input variables is a mapping $f: \{0, 1\}^n \rightarrow \{0, 1, *\}$. Each element of $\{0, 1\}^n$ is called a *vertex* in the n–dimensional Boolean cube.

For all vertices, if *f* evaluates to {0,1} then *f* is a *completely specified logic function*, otherwise *f* is a *incompletely specified logic function*.

The *on–se*t of *f* is defined as the set of vertices for which *f* evaluates to 1; the off–set, the set of vertices for which *f* evaluates 0; the don't–care set, the set of vertices for which *f* evaluates to * (i.e not specified). Each vertex of the on–set of *f* is called *minterm* and that of the off–set of *f* is called *maxterm*.

A *literal* is a variable or its complement. A cube *c* is a set of literals, such that if $a \in c$ then $\overline{a} \notin c$ and vice–versa. Each vertex of an n–dimensional cube is represented by a cube with n literals.

A cube is expanded by removing literals. When expanded, the resultant cubes cover more vertices or more cubes with higher number of literals. A cube *c'* covers another cube *c"*, if $c' \subseteq c''$, for example $\{\overline{a}, b\}$ covers both $\{\overline{a}, b, d\}$ and $\{\overline{a}, b, \overline{d}\}$.

An on–set cover *F* of a logic function *f* is a set of cubes such that each minterm of *f* is covered by at least one cube of *F*. The on–set cover is the sum–of–product (SOP) implementation of the function *f*. Similarly an off–set cover *R* of a logic function *f* is a set of cubes such that each maxterm of *f* is covered by at least one cube of *R*. The off–set cover is the product–of–sum (POS) implementation of the function *f*.

A cube in an on–set (off–set) can be expanded by covering all the other cubes in the on–set (off–set) and the cubes in the don't–care set. A cube is called prime implicant of *f* if the cube cannot be expanded further. A cover is called prime cover of a function *f* if all of its cubes are the prime implicants of *f*. A cover is called irredundant if it ceases to be a cover after removing any one of its cubes.

A function *f* is monotone increasing in a variable x_i if
$f(x_1, x_2, \dots, x_i=0, \dots x_n) = 1 \Rightarrow f(x_1, x_2, \dots, x_i=1, \dots x_n) = 1$,
that is increasing the value of x_i from 0 to 1 never decreases the value of *f* from 1 to 0.

A function *f* is monotone decreasing in a variable x_i if
$f(x_1, x_2, \dots, x_i=1, \dots x_n) = 0 \Rightarrow f(x_1, x_2, \dots, x_i=0, \dots x_n) = 0$,
that is decreasing the value of x_i from 1 to 0 never increases the value of *f* from 0 to 1.

A function f is positive (negative) unate in variable x_i if it is monotone increasing (decreasing) in variable x_i . A function f is binate if it is neither positive nor negative unate.

A variable x_i is said to have controlling value in a function f, if
$f(x_1, x_2, \dots, x_i=1, \dots x_n) = 0(1) \Rightarrow f(x_1, x_2, \dots, x_i=0, \dots x_n) = 1(0)$
i.e. $\frac{df_x}{dx_i} = f(x_1, x_2, \dots, x_i, \dots, x_n) \oplus f(x_1, x_2, \dots, \overline{x_i}, \dots, x_n) = 1$

If the variable x_i has a controlling value in a function f, then the function f is binate in the variable x_i.

3 Syntactic Constraints

STGs are behavioural specifications of SGs from which state graphs can be derived to realise the final control circuit. STGs are more concise, because they do not require a large number of states to describe concurrent occurrences of control events, in contrast to the case of SGs. In order to realise deadlock–free and hazard–free circuits, SGs should satisfy certain properties [3, 7, 13].

The properties of SGs can be redefined as the corresponding _syntactic constraints_ of STGs because, they can be checked and modified more easily in STGs due to the fewer number of transitions than the number of states of SGs.

3.1 STG liveness

In the process of obtaining a logic function from an STG specification, the first step is to obtain the state graph with consistent state assignment (Section 2.2.1). From the concept of consistent state assignment it can be deduced that, in every simple cycle of an SG, the number of positive (_t_+) and negative (_t_–) transitions must be equal, and they must alternate. If the STG is live (Section 2.1.4), then every pair of positive and negative transitions of a signal are ordered. Thus, the STG liveness is a necessary condition to implement a circuit from an STG specification.

3.2 Persistency

Persistency is one of the important properties of signal transition graphs. A Petri net is said to be persistent if, for any two enabled transitions, the firing of one transition will not disable the other [14]. In state graphs, persistency is referred to as _semi–modularity_ [12]. Using an informal definition from [12], in a semi–modular state graph, if a signal transition _t_+ (or _t_–) is excited in state _a_, but the signal does not change to 0 or 1 (from 1 or 0) when the circuit goes to a new state _b_, then the signal must still be excited in state _b_ and have the same value as in state _a_ . In [12], it was proved that semi–modularity is a sufficient condition for speed–independence with respect to all the signals in a state graph.

3.2.1 Chu's STG Persistency

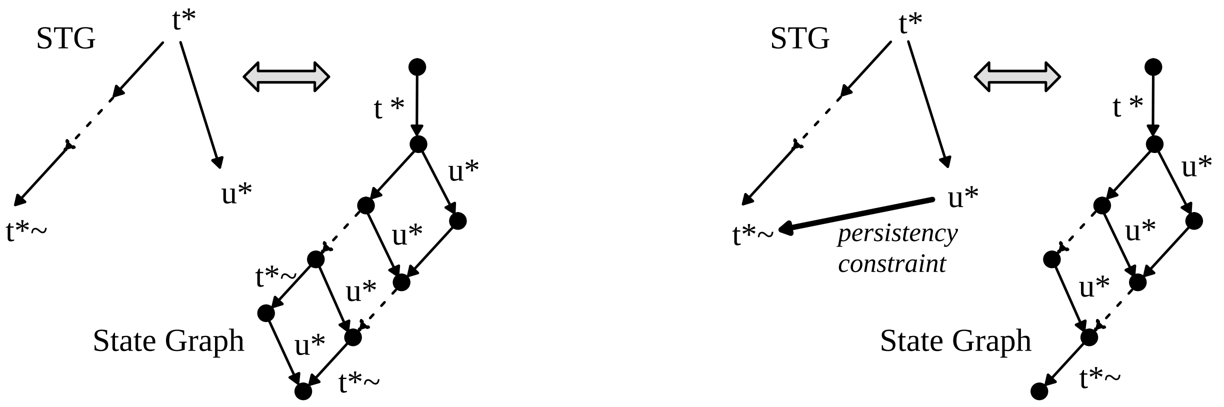
In an attempt to characterize speed–independence at the STG level, Chu [3] introduced the STG persistency constraint as an equivalent to persistency in state graphs. According to Chu, an STG is called persistent if all of its non–input transitions are persistent. A transition _u_ is non–persistent if transition _t_* enables _u_, and _u_ and _t_*~, the complementary transition of _t_*, are concurrent as shown in Figure 2(a). Persistency is justified as a necessary syntactic constraint by considering the corresponding logic implementation in the case of non–persistency. In Figure 2(a), concurrency between _u_ and _t_*~ implies that while the logic element is reacting to _t_* to cause _u_, _t_*~ may be occurring simultaneously at the input of that logic element. This represents a race condition for the circuit. This race can be eliminated by introducing a persistency constraint into the graph.

A _persistency constraint_ is an ordering constraint between two transitions, namely from _u_ to _t_*~ (_u_ $\xrightarrow{*}$ _t_*~) as shown in Figure 2(b). Note that the addition of the ordering constraint does not change the behaviour of the system, but may reduce the level of concurrency in the specification. Chu assumed that the transitions of all _input_ signals are always persistent. The reason for this assumption is that even if two transitions of input signals appear to be enabled in the same state, in a global system comprising the original system and the environment, they may indeed be enabled in different state. Thus Chu's persistency is defined as follows

> An STG is persistent if and only if for every non–input signal _j_, a transition of _j_ caused by a transition _t_* is ordered with _t_*~.

3.3 STG Persistency versus Consistent State Coding (CSC)

Moon [13] proved that a state graph satisfies CSC property if and only if it is semi–modular. Transformations can be applied at the STG level to enforce the CSC property of the state graphs.



(a) Non–persistent transition u*

(b) Persistent transition u*

Figure 2. Illustration of Chu's STG persistency constraint [2] definition

A CSC violation can be corrected either by inserting an extra signal in the STG so as to distinguish the states violating CSC, or by introducing a new ordering constraint between the signal transitions so as to remove some of the states violating CSC. An algorithm for solving the CSC problem in STGs was developed in [17, 18]. The STG persistency constraint is also used by a number of researchers to satisfy CSC [3, 10] because it is a special case of solving the CSC problem. Thus the algorithm presented in [10] which is based on STG persistency constraints may add more constraints than are necessary for consistent state assignment, as observed in [17].

3.3.1 The Argument against Chu's STG Persistency

Let us consider the non–persistent STG given in Figure 1(c). The transition *y+* is non–persistent because the transitions *x+* and *y+* are ordered (i.e *x+* → *y+*) and the transitions *y+* and *x–* are concurrent. The condition when x– disables y+ results in a hazard according to the persistency concept. But the corresponding state graph in Figure 1(d) satisfies CSC because all the states are represented by a unique binary vector. The CSC property ensures hazard–free implementation. The hazards at this stage mean undesirable behaviour (not according to the initial specification) at the functional level. Thus, this example shows that the STG persistency constraint is not a necessary condition to satisfy CSC.

In Figure 1, the logic equation for the non–persistent signal *y* is *y=x+z*. When *x–* fires *z* is already at 1, due to the causal relation *z+* → *x–*. Since *z* is a controlling value for the *or* gate, *x–* cannot disable *y+*. The concept of *controlling value* was introduced by Luciano in [8]. Recently Puri [15] has proved the following theorems

> **Theorem 1** : If there is a signal transition *s** between *t** and *t*~*, such that *s* has a controlling value in *u* , then non–persistency of *u* neither causes a hazard nor a CSC violation (refer Figure 2).

> **Theorem 2** : If there is a signal transition *s** between *t** and *t*~* such that s is a non–controlling value in *u*, then non–persistency of *u* must cause a CSC violation (refer Figure 2).

The above theorems do not provide a visual and intuitive relationship between persistency and CSC, because we will know whether a signal *s* has a controlling value (according to the definition in Section 2.4) in a non–persistent signal *u* only after deriving the logic. If an STG satisfies CSC property, all the signal transitions in the STG must be *literally* persistent.

According to Chu [3], the persistency of a non–input transition *u** is defined solely based on a transition *t** with a causal relation *t** → *u** and the relationship between the complementary transition *t*~* and *u**, and does not consider the relationship between all other signal (except *t*) transitions and *u**. Thus, we consider the persistency constraint introduced by Chu as *local STG persistency constraint* and introduce a *global STG persistency concept* considering all the signal transitions in the STG.

In the following Section, we show that whenever there is a CSC violation in an STG, a signal is non–persistent(literally). The global persistency definition represents the *literal* persistency of all the signals and has a one to one correspondence with the CSC property of state graphs. From now on persistency refers to the *literal* persistency or global persistency. We also show that the persistency constraint of Chu is a special case of global persistency constraints. Section 5 discusses the conditions under which a signal has a controlling value in another signal.

4 Global STG Persistency

Complete state coding is the only necessary and sufficient condition for implementing speed–independent, hazard–free circuits. The global persistency property of STGs is exactly the same as the CSC property of state graphs. An STG is persistent if all the non–input transitions are persistent with respect to all other signal transitions in the STG. The conditions at which a signal transition is persistent or non–persistent with respect to another signal transition are discussed below.

4.1 Non–persistent Transition *u** with respect to a Transition *p**

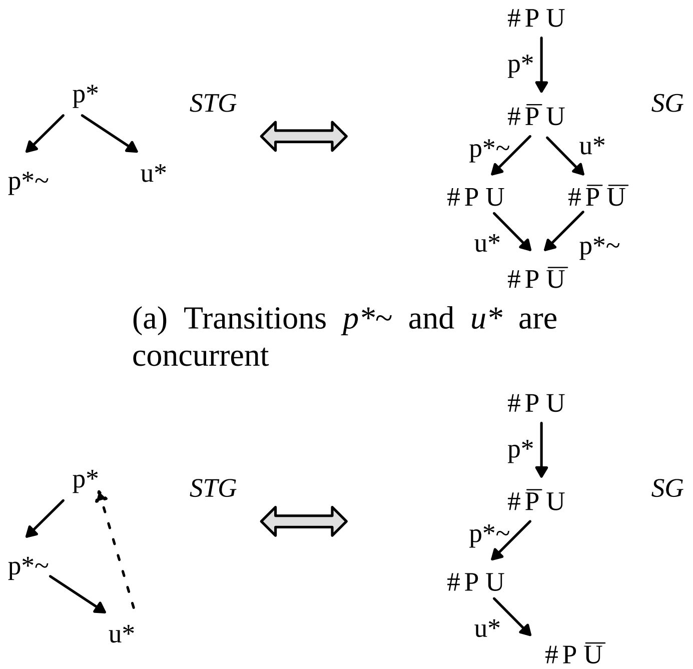
The different causal constraints that may cause a signal transition to be non–persistent are considered. We present direct proof that relates those causal constraints and the CSC violation. Let us consider an STG in which a signal transition *u** is locally non–persistent (according to Chu's definition [3]), i.e a transition *p** enables *u** and the complementary transition *p*~* and *u** are concurrent (Figure 3 and Figure 4).

4.1.1 No Signal Transition between Transitions *p** and *p*~*

Let us consider an STG where there is no signal transition between transitions *p** and *p*~* as shown in Figure 3(a). The signal transition *u** is non–persistent (locally and globally) if it is ordered with *p** and concurrent with *p*~*. The definitions for local and global persistencies are the same in this case. The formal definition of global persistency is given in the next section. The following theorem [15] establishes that the non–persistency in this case always cause a CSC violation in the SG .

Theorem 3 : *If there is no signal transition between* p* *and* p*~*, and the transition* u* *is ordered with* p* *and concurrent with* p*~*, then the signal* u *is non–persistent and will cause a CSC violation in the state graph.*

Proof : Let the binary values of *p* and *u* before firing the transition *p** be *P*, *U*. Let the values of the remaining input signals to the sub–circuit implementing the signal *u* be represented as *#* (because they are not affected by the transitions of *p* and *u*). Thus the binary coding of the state in which *p** is enabled is equal to *#PU*. A signal transition after its firing will change the signal value in the opposite direction of its



(a) Transitions $p*\sim$ and $u*$ are concurrent

(b) Transitions $p*\sim$ and $u*$ are ordered with $p* \dashrightarrow u*$ – *u* is persistent (according Chu's definition)

Figure 3. Non–persistent signal *u* with respect to signal *p* when no signal transition is between $p*$ and $p*\sim$

current value. The resultant SG of the STG where the signal *u* is non–persistent is given in Figure 3(a). It can be observed that the SG has two states with the same binary code *#PU* in which the two different transitions $p*$ and $u*$ are enabled. It is also evident that from a particular state, after executing the sequence $p* \rightarrow p*\sim$, we will reach another state with same binary code. This condition says that the state graph has CSC violation. The same binary code indicates that the two states in fact represent a single state enabling two transitions $p*$ and $u*$. The transition $u*$ may not be persistent depending on the delay of firing $p*$. Since the persistency of $u*$ is depend on the delay of firing $p*$, we consider *u* as non–persistent with respect to *p*. Thus if there is no signal transition between $p*$ and $p*\sim$, and the transition $u*$ is ordered with $p*$ and concurrent with $p*\sim$, then *u* is non–persistent and causes CSC violation. ■

Let us now consider the STG given in Figure 3(b), where the transition $u*$ is ordered with $p*\sim$ and not concurrent with $p*$ (i.e. $p* \rightarrow p*\sim \rightarrow u*$). The following theorem establishes that Chu's persistency constraint is not sufficient to ensure CSC property.

Theorem 4 : *If there is no signal transition between* p* *and* p*~ *and the transition* u* *is ordered with* p*~ *and not concurrent with* p* *(i.e.* p* → p*~ → u**), then the signal* u *which is persistent according to Chu's definition (actually non–persistent globally – refer Definitions 1&2) will cause a CSC violation in the state graph.*

Proof : The relevant SG is given in Figure 3(b). It can be easily seen that the SG has two states with the same binary code and violates CSC. In fact the causal constraint $p* \rightarrow p*\sim \rightarrow u*$, which is a particular case of the concurrent relationship between $p*\sim$ and $u*$ (i.e either $p* \rightarrow p*\sim \rightarrow u*$ or $p* \rightarrow u* \rightarrow p*\sim$ – refer Figure 3(a)), causes

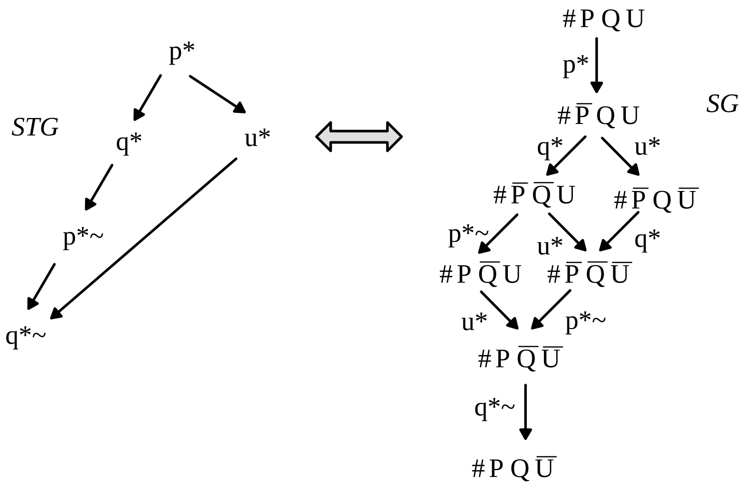

STG
p*
q*
u*
p*~
q*~
SG
# P Q U
p*
# $\overline{P}$ Q U
q*
u*
# $\overline{P}$ $\overline{Q}$ U
# $\overline{P}$ Q $\overline{U}$
p*~
u*
q*
# P $\overline{Q}$ U
# $\overline{P}$ $\overline{Q}$ $\overline{U}$
u*
p*~
# P $\overline{Q}$ $\overline{U}$
q*~
# P Q $\overline{U}$

Figure 4. Non–persistent *u* (according to Chu's definition) with no CSC violation – actually *u* is persistent

the CSC violation and the signal *u* to be non–persistent. But according to Chu's persistency definition[3], the signal *u* is persistent even though it is actually non–persistent. Chu[3] dealt with this problem separately with consistent state assignment. Thus this example proves that the Chu's persistency constraint is not sufficient to ensure CSC property. ■

4.1.2 Signal Transition between Transitions *p** and *p*~*

Figure 4 shows an STG, in which there is a signal transition q* between transitions p* and p*~. The signal transition u* is ordered with p* and concurrent with q* and p*~. The following theorem proves that Chu's persistency constraint is not a necessary condition to ensure CSC violation.

Theorem 5 : *A non–persistent signal according to Chu's definition may actually be persistent (refer Definition 3 for global persistency) and may not cause CSC violation.*

Proof : According to Chu's definition [3], the signal *u* in Figure 4, is non–persistent (locally). Let the binary values of *p*, *q*, *u* before the transition *p** fires be *P*, *Q*, *U* respectively and let the values of the remaining signals be denoted by # (because they do not change in the transition sequence of Figure 4). Figure 4 also shows the corresponding SG derived from the STG. It can be seen that there are no states having same binary code and thus there is no CSC violation in the state graph. This example shows that Chu's persistency constraint is not necessary to ensure the CSC property in state graphs. ■

Thus theorems 3&4 show that Chu's persistency constraint is neither a necessary nor a sufficient condition to ensure the CSC property. The causal constraints between *p**, *p*~*, *q** and *u** are not sufficient to decide whether the signal *u* is globally persistent or not. The global persistency definition requires also the causal constraint between *q*~* and *u** and is defined in general as follows

Definition 1 *: If there is no signal transition between a sequence of transitions,* s* = {t_1*, t_2*, . . . , t_i*, . . . , t_n*} *and a sequence of complementary transitions,* s*~ =

{t_1*~, t_2*~, . . . , t_i*~, . . . , t_n*~} *and if the signal transition* u* *is ordered with one of the transitions of* s* *or* s*~*(i.e* t_i* → u* *or* t_i*~ → u**), then the signal* u *is non–persistent with respect to signal* t_1 *(refer Figure 5).*

***Definition 2** : An STG is globally non–persistent, if any of its non–input signals is non–persistent with respect to any of the other signals in the STG.*

In the above definitions, the transitions in the sequences s* *and* s*~ *can be concurrent. If two transitions* t_1* *and* t_2* *are concurrent, they can be viewed as either* t_1* → t_2* *or* t_2* → t_1*.

When *s** = *p** and *s**~ = *p**~, then the signal *u* is non–persistent and causes a CSC violation in the state graph as explained in Section 4.1.1. Figure 5 illustrates the different conditions in which the signal *u* is non–persistent with respect to *p**, when *s** = {*p**, *q**} and s*~ = {*p**~, *q**~}. The following theorem establishes a relation between the non–persistency of a signal with respect to another signal and CSC violation.

Theorem 6 : *A signal in an STG which is non–persistent with respect to another signal (refer Definition 1) causes a CSC violation in the state graph and vice–versa.*

Proof : (⇒) Let *n* be the number of transitions in *s** and *s**~. When there is a single transition (*n=1*) in the sequences *s** and *s**~, the non–persistent signal *u* will cause a CSC violation and this condition is proven under Theorems 3&4. Let us consider transitions of two signals *p* and *q* in the sequences *s** and *s**~ as shown in Figure 5. The different ways in which the transition *u** is ordered are shown in case (1) to case (4) of Figure 5. In Case (1), *u** is ordered with *p** and concurrent with all the remaining transitions whereas in Case(4), *u** is not concurrent with any of the signal transitions. Since there is no signal transition between *s** and *s**~, the sequences of transitions *s** and *s**~ can be grouped as single transitions *t** and *t**~ irrespective of the number of transitions in *s** and *s**~. The resultant simplified STGs have transitions *t** and *t**~ with no signal transition in between them and *u** is either concurrent or ordered with *t**~ as shown in Figure 5. Theorems 3 & 4 say that the simplified STGs will cause a CSC violation in the state graph. Thus, a non–persistent signal u always causes a CSC violation in the state graphs irrespective of the number of transitions in the sequences s* and s*~. The SGs corresponding to the simplified STGs (with transitions of *t* and *u*) and the final SGs corresponding to the original STGs (with transitions of *p*, *q* and *u*) are also given in Figure 5. The SGs also illustrates that there is CSC violation in the state graphs. The final SGs in Figure 5 are not complete but give enough information to prove a CSC violation. All the final SGs have two states with same binary code where the transitions *p** and *u** are enabled. The same binary code indicates that the two states in fact represent a single state enabling two transitions *p** and *u**. The transition *u** may not be persistent depending on the delay in firing *p**. This clearly gives rise to a malfunction or hazard. Since the persistency of *u** is depend on the delay in firing *p**, the signal *u* is defined to be non–persistent with respect to *p** (first signal transition in *s**). Thus a non–persistent signal must always cause a CSC violation.

(⇐) Assume the state graph has CSC violation, i.e it has two states with same binary code enabling two different signal transitions. In a state graph, two states will have the same binary code if and only if both transitions of the signals (*t** and *t**~) occur between them as shown in Figures 3&5. The STGs corresponding to the SGs with CSC violation are also given. It can be observed that there is no single signal

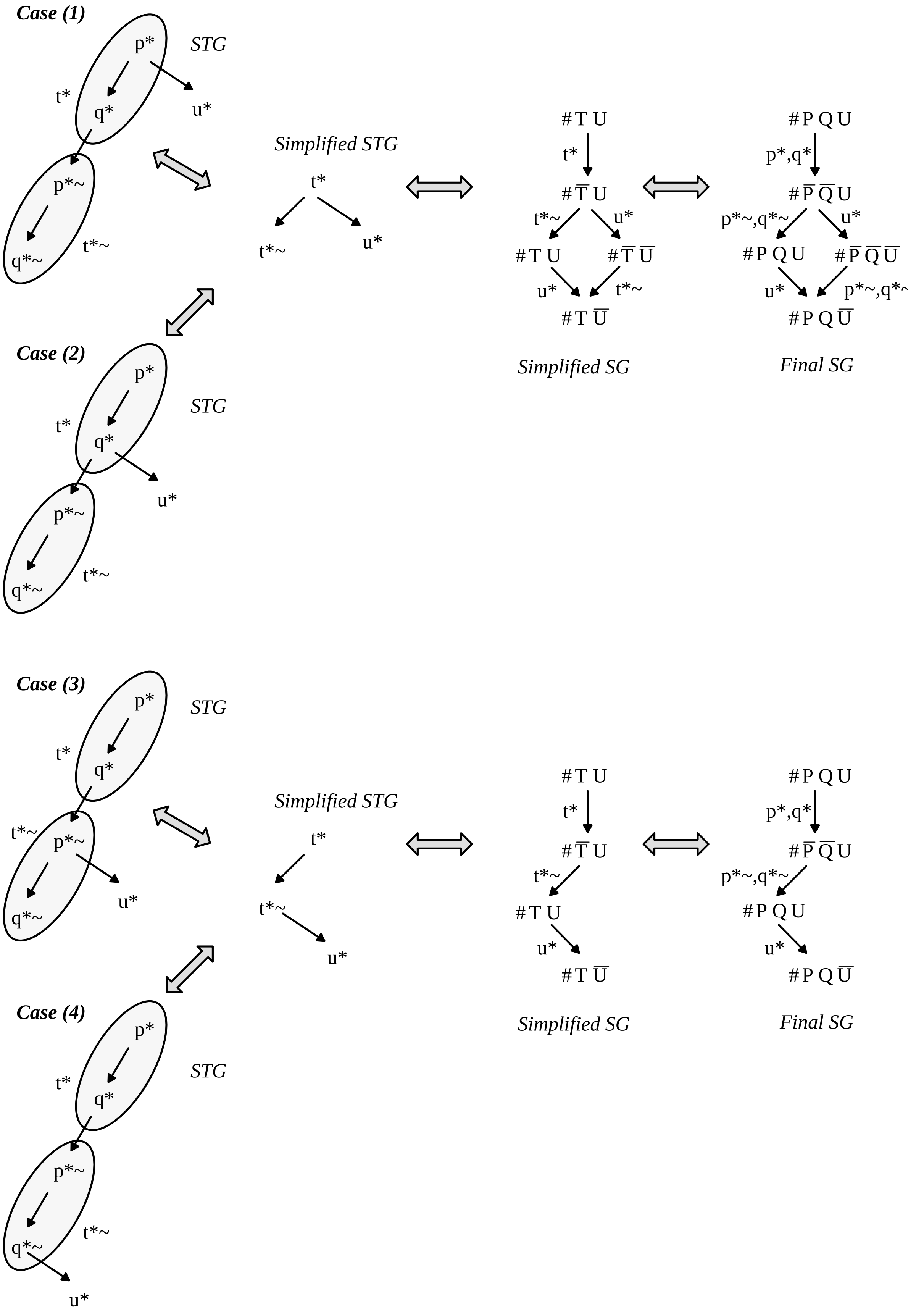


Figure 5. Non–persistent signal *u* with respect to *p* when $p^* \to q^* \to p^{*\sim} \to q^{*\sim}$

transition between a sequence of transitions (*p**, *q** in Figure 5) and a sequence of the corresponding complementary transitions (*p**~, *q**~ in Figure 5) and there is a transition (u*) ordered with one of the transitions of the sequences. Thus the signal *u* is non–persistent with respect to the signal *p* whose transition (*p** in Figure 5) is first among the sequence of transitions. ■

4.2 Global Persistency Constraint

In this Section, we introduce the persistency definition and the constraints that can be used to enforce CSC property in state graphs.

***Definition 3** : An STG is persistent if for all signals, there is a transition in between the transitions of the sequences* s* = {t_1*, t_2*, . . . , t_i*, . . . , t_n*} *and* s*~ = {t_1*~, t_2*~, . . . , t_i*~, . . . , t_n*~}. *An STG is also persistent even if there is no transition between* s* *and* s*~*, but there is no other transition u* which is ordered with one of the transitions of s* and s*~.*

The persistency definition is exactly opposite of the non–persistency definition. Thus as a direct consequence of Theorem 6, we can derive a corollary about the relationship between persistent STG and CSC property of the corresponding state graph.

Corollary : *A persistent STG always satisfies the CSC property in the corresponding state graph.*

Proof : In a persistent STG, since there is a transition between the transitions of *s** and *s**~, all the transitions of *s** and *s**~ do not occur consecutively and do not produce two states with the same code. According to the Theorem 6, the state graph with CSC violation must have a corresponding STG with a non–persistent transition. Thus a persistent STG always satisfies the CSC property in the corresponding state graph. ■

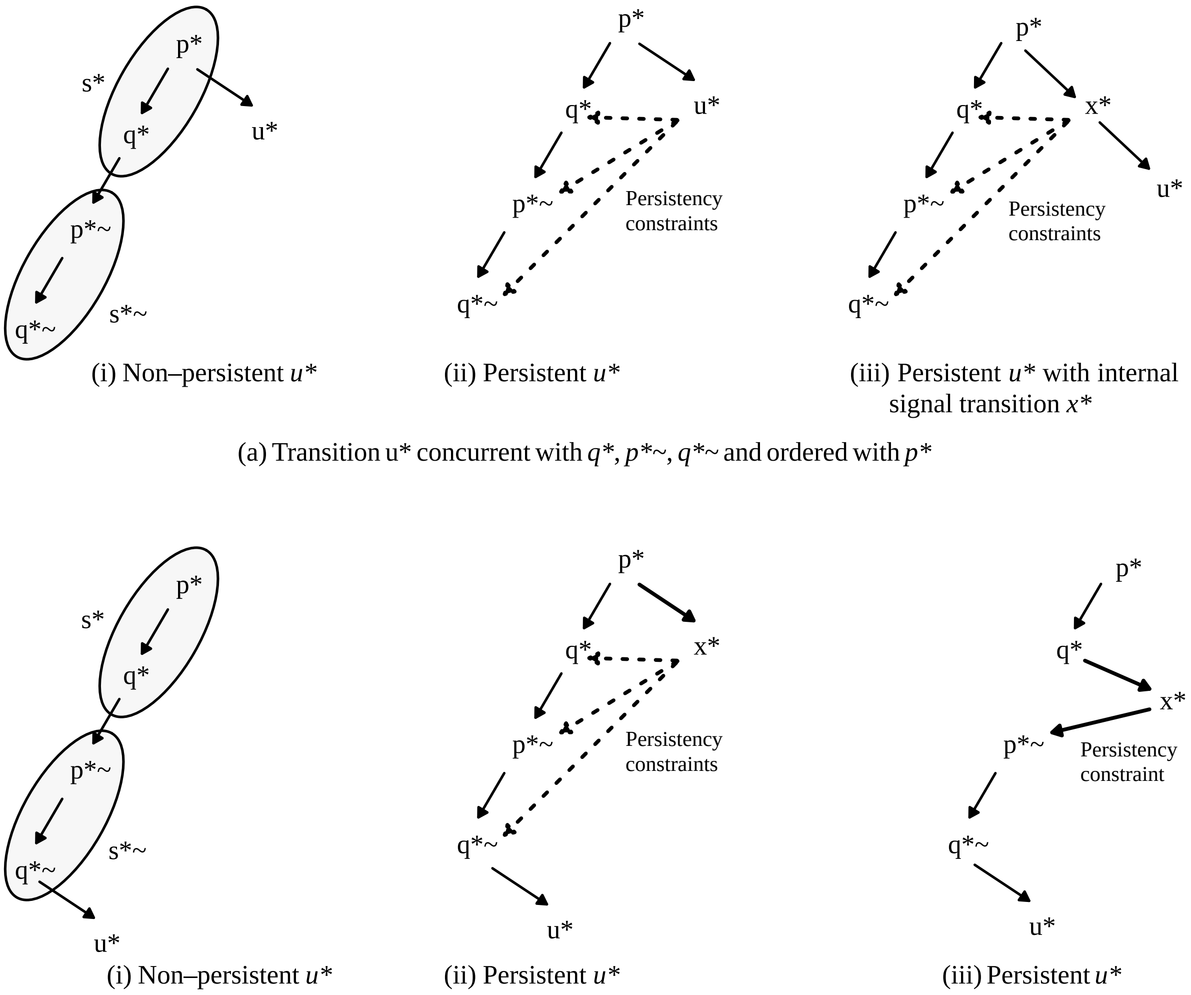
The definitions and the proofs relate to the persistency assumes that every signal in the STG has exactly 1 up and down transition. The persistency definition also supports signals in an STG having multiple up and down transitions as long as there is at least one signal transition between multiple occurrences of the same signal transitions. Let *s** = {t_1*, t_2*, . . . , t_i*, . . . , t_n*} be a sequence of transitions and *s**~= {t_1*~, t_2*~, . . . , t_i*~, . . . , t_n*~} be the corresponding sequence of complementary transitions.

Restriction 1 : *The global persistency concept supports an STG having multiple occurrences of* s* *and* s*~ *as long as there is a signal transition* x* *between the two occurrences* s* *or* s*~.

The reason for the above restriction is that the resultant state graph has two states with same binary code enabling two signal transitions t_1* which need to be differentiated. Otherwise there will be a CSC violation in the state graphs. If there is no signal transition between the multiple occurrences *s** and *s**~ then an internal signal transition *x** can be introduced between two occurrences of *s** or *s**~ such that *x** and *x**~ satisfies STG liveness.

A non–persistent signal can be made persistent in a number of ways. The persistency constraint introduced by Chu [3] is one way of enforcing persistency of a signal. The following definition gives a number of ways in which a non–persistent signal can be made persistent.

***Definition 4** : A non–persistent transition* u* *with respect to* t_1* *(refer Definition 1) can be made persistent by (a) introducing a constraint such that* u* *and* t_n*~ *are*



(i) Non–persistent *u**   (ii) Persistent *u**   (iii) Persistent *u** with internal signal transition *x**

(a) Transition u* concurrent with *q*, p*~, q*~* and ordered with *p**

(i) Non–persistent *u**   (ii) Persistent *u**   (iii) Persistent *u**

(b) Transition u* ordered with *p*, q*, p*~* and *q*~*

Figure 6. Different constraints to enforce persistency

> *ordered (i.e if* $t_i^* \rightarrow u^*$*, the constraint is* $u^* \overset{*}{\rightarrow} y^* \in \{t_{i+1}^*, \ldots, t_i^*, t_{1^*\sim} \ldots, t_n^{*\sim}\}$ *and if* $t_i^{*\sim} \rightarrow u^*$*, the constraint is* $u^* \overset{*}{\rightarrow} y^* \in \{t_{i+1}^{*\sim}, \ldots, t_n^{*\sim}\}$, y^* *being a non–input signal transition), or (b) introducing an extra internal signal transition* x^* *such that either* $t_i^* \rightarrow x^* \rightarrow u^*$ *or* $t_i^{*\sim} \rightarrow x^* \rightarrow u^*$ *and* x^* *and* $t_n^{*\sim}$ *are ordered, or (c) introducing an extra internal signal transition* x^* *between transitions of* s^* *or* $s^{*\sim}$*, but not before a non–input signal transition. The internal signal transition* x^* *is introduced in such a way that* x^* *and* $x^{*\sim}$ *does not cause further non–persistency and satisfies STG liveness.*

One of the three conditions given in the above definition can be used to enforce persistency in a non–persistent STG. If the non–persistent transition *u** is ordered with $t_n^{*\sim}$ (i.e $t_n^{*\sim} \rightarrow u^*$) then the non–persistency can be removed by inserting an internal signal transition. Figure 6 illustrates the different ways in which persistency can be enforced. The non–persistent *u** in Figure 6(a)(i) can be made persistent in two different ways as shown in (ii) and (iii) of Figure 6(a). In Figure 6(a)(ii), the persistency constraint reduces the level of concurrency in the description where as in Figure 6(a)(iii) the internal signal transition *x** is introduced to maintain the concurrency between *u** and *(q*, p*~, q*~)*. The constraints $u^* \rightarrow p^{*\sim}$ and $x^* \rightarrow p^{*\sim}$ are same as the persistency constraints defined by Chu[3]. Since the non–persistent *u** is ordered with all the transitions in Figure 6(b)(i), it can be made persistent only by introducing an internal signal transition *x** between *p** and *q*~*. The transition x* can be introduced in a number of configurations and two typical configurations are given in Figure 6(b) (ii) & (iii). The dotted edges represent many possibilities.

5 Relationship between Controlling Value and CSC

In Section 3.3.1, we discussed the relation between a signal having a controlling value in another signal and the CSC. Puri [15] has proved Theorems 1 & 2 that relate the controlling value, CSC property and CSC violation. The formal definition of controlling value is given in Section 2.4. In this Section, we provide STG syntactic constraints in order to determine whether a signal has a controlling value in another signal. If a signal *x* has a controlling value in a non–input signal *y* then *x* can be classified either as a *trigger signal* or as a *context signal* [18,19].

5.1 Trigger Signals

A signal *x* is called *trigger signal* for a non–input signal *y* if a transition of the signal *x* causes the signal *y* to change immediately. Obviously, such a trigger signal *x* is an input signal for the logic generating *y*. *x* is a trigger signal of *y* iff there is an arc in the STG going from a transition of *x* to a transition of *y* ($x* \rightarrow y*$), provided that the arc is not redundant. An arc from *x** to *y** is redundant if there is another path from *x** to *y** with a signal transition in between them. In Figure 7(a) [18], signals *A* and *D* are the trigger signals to the signal *B* because of the causal relations $a+ \rightarrow b+$ and $d+ \rightarrow b-$.

5.2 Context Signals

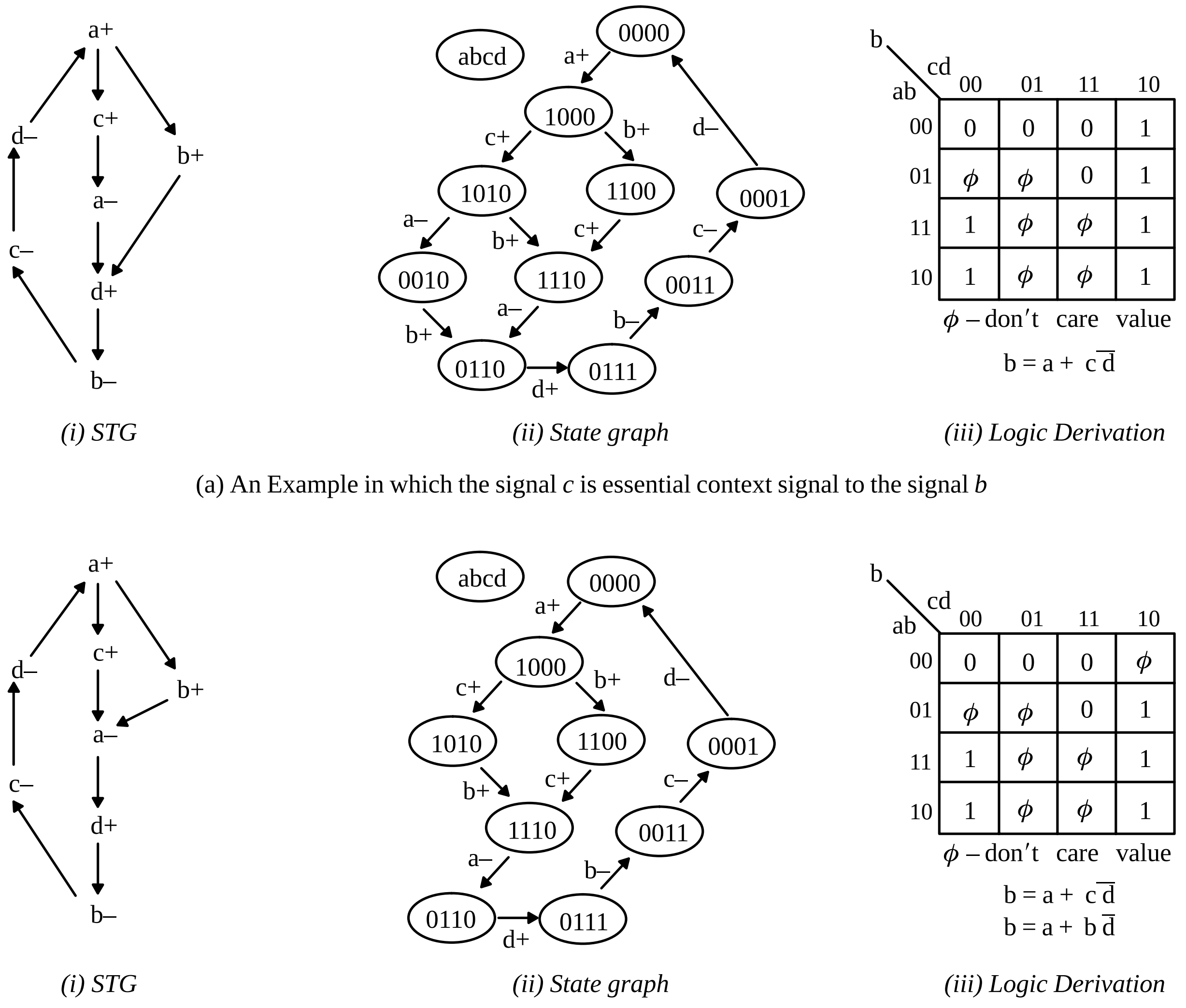
A signal *x* is called *context signal* for a non–input signal *y* if *x* is not a trigger signal of *y* but is an input signal for the logic generating *y*. In an STG, context signals do not have direct arcs between their transitions and the transitions of the non–input signals that are being implemented. Context signals may vary according to the logic implementation, and can be further classified into two categories, *essential context signals* and *non–essential context signals*.

The essential context signals are the input signals that are necessary to the logic being implemented, i.e if the transitions of the essential context signals are removed from the STG, then the STG becomes non–persistent and has a state graph with CSC violation. An essential context signal is defined formally as follows.

> ***Definition 5*** *: A signal* x *is called an essential context signal to a non–input signal* y *if* x *is not a trigger signal and if the persistent signal* y *becomes non–persistent by removing the transitions of the signal* x *from the STG.*

Non–essential context signals are signals that are not essential but can be used to implement the logic i.e. there is always a logic realisation with out non–essential context signals. In Figure 7(a), *c* is the essential context signal to the signal *b*, where as in Figure 7(b), *c* is the non–essential context signal to the signal *b*. The corresponding logic realisations are also given in Figures 7(a)&7(b). It can be seen in Figure 7(b), that the signal *b* has a realisation without *c* as an input signal. The following theorem gives the relation between the context signals and the global persistency of an STG.

> **Theorem 7** : *The signal* x *whose transition* x* *(either* x+ *or* x– *but not both) is concurrent with a non–input signal transition* y* *is the context signal to the signal* y *iff the STG is persistent.*



(i) STG

(ii) State graph

(iii) Logic Derivation

ab \ cd	00	01	11	10
00	0	0	0	1
01	ϕ	ϕ	0	1
11	1	ϕ	ϕ	1
10	1	ϕ	ϕ	1

ϕ – don′t care value

$b = a + c\,\overline{d}$

(a) An Example in which the signal *c* is essential context signal to the signal *b*

(i) STG

(ii) State graph

(iii) Logic Derivation

ab \ cd	00	01	11	10
00	0	0	0	ϕ
01	ϕ	ϕ	0	1
11	1	ϕ	ϕ	1
10	1	ϕ	ϕ	1

ϕ – don′t care value

$b = a + c\,\overline{d}$

$b = a + b\,\overline{d}$

(b) An Example in which the signal *c* is non–essential context signal to the signal *b*

Figure 7. Examples illustrating the trigger and different context signals

Proof : ($\Rightarrow$) Assume the STG is persistent. According to the Corollary given in Section 4.2, a persistent STG always has a state graph with the CSC property. Let us consider the persistent STG and the state graph shown in in Figure 7(a). The transition *c+* is concurrent with the non–input signal *b+*. It can be observed from the state graph that there is an implicit constraint $c+ \rightarrow b+$ i.e, *c* has controlling value in *b* at a particular vertex of boolean cube. The persistency of the STG assures that all the constraints (explicit and implicit) are persistent. Thus the signal *c* is the context signal to the signal *b*. In general, if a signal transition *x** is concurrent with another transition *y** in a persistent STG then there will be an implicit constraint $x* \rightarrow y*$ indicating that *x* is context signal to the signal *y*. If signal *x* has both of its transitions *x+* and *x–* concurrent with *y** then the state graph will have two implicit constraints $x+ \rightarrow y*$ and $x– \rightarrow y*$ indicating that the signal *x* is redundant with respect to the signal *y*.

($\Leftarrow$) Assume the STG is non–persistent. According to Theorem 6, the corresponding state graph of the STG has a CSC violation. Let us consider a non–persistent STG given by case(1) of Figure 5. The corresponding state graph has an implicit constraint $q* \rightarrow u*$. The constraint $q* \rightarrow u*$ may not be persistent because the circuit can have the sequence of transitions $p* \rightarrow q* \rightarrow p*\sim \rightarrow q*\sim$ an infinite number of times with out *u** occurring. This indicates a hazard condition for the signal *u*. The signal *q* does not have any control on the signal *u* indicating that *q* is not a context signal to the signal *u*. Thus, the signals whose transitions are concurrent with a transition in a non–input signal, are not context signals to that non–input signal, if the given STG is non–persistent. ■

The following theorem gives a relationship between an essential context signal and trigger signals. Before stating the theorem we need the following definitions for interleaving [18].

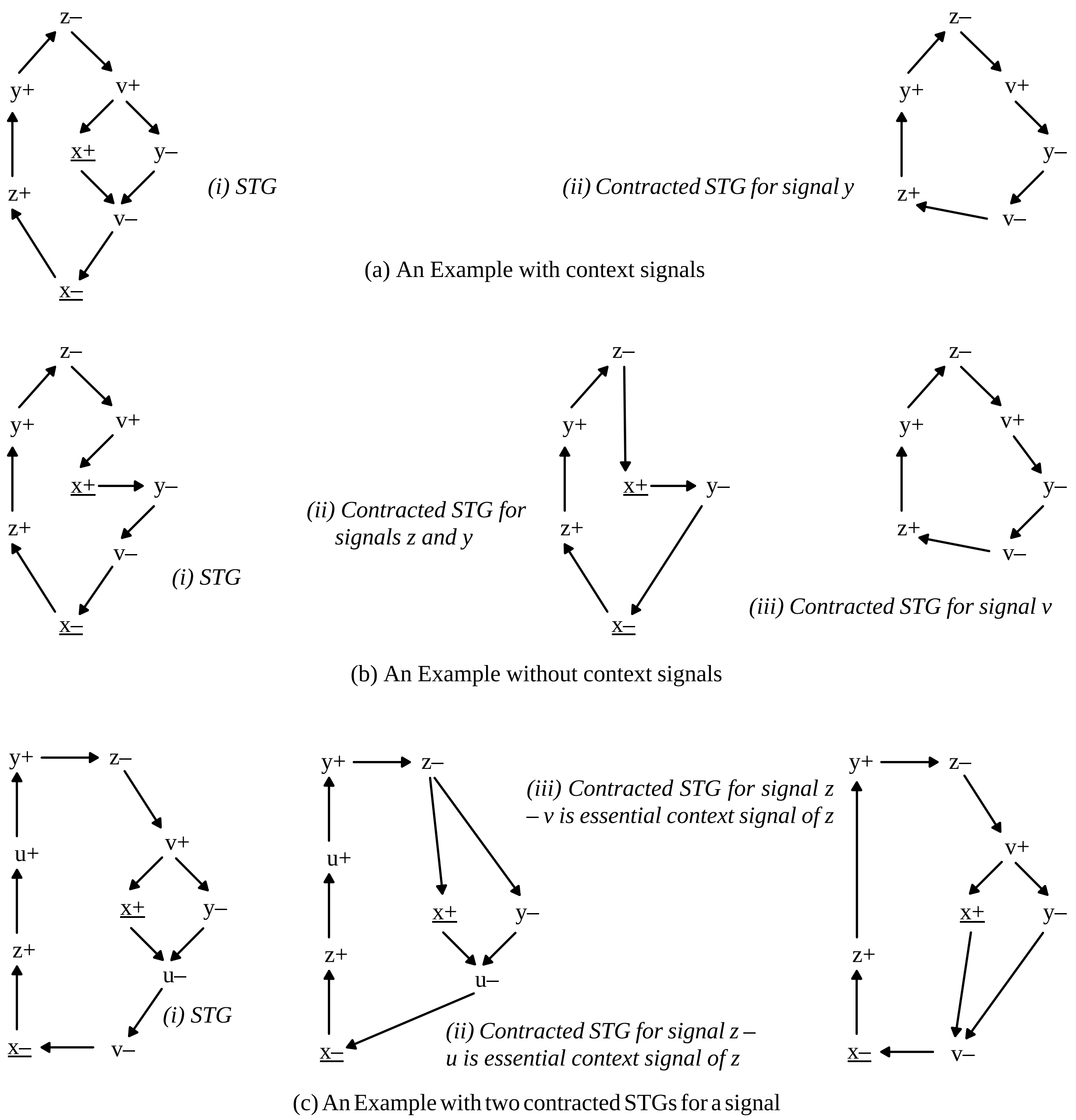
Definition 6 : *Transitions of a signal* x *are said to be interleaved with the transitions of a signal* y*, denoted by* I(x*, x*~, y*, y*~)*, in an STG iff there exists a simple cycle on which the transitions are ordered as* $x* \xrightarrow{*} y* \xrightarrow{*} x*\sim \xrightarrow{*} y*\sim$.

Theorem 8 : *Let the signals* x *and* y *be the trigger signals of a non–input signal* z*. If both the transitions of a trigger signal* x *occur between the transitions of* z *(i.e. transitions of signals* x *and* z *are not interleaved) and transitions of both the trigger signals are not interleaved then, the transitions of the essential context signal must be interleaved with the transitions of the trigger signal* x *and with the transitions of either the trigger signal* y *or the non–input signal* z*.*

Proof : Let us consider the STG shown in Figure 8(a). The signals *x* and *y* are the trigger signals to the signal *z* because of the constraints $x- \rightarrow z+$ and $y+ \rightarrow z-$. The transitions *x+* and *x–* occur between *z–* and *z+*. It can be observed that the signal *v* has the transition *v–* between *x+* and *x–*, and the other transition *v+* between *y+* and *y–* satisfying the above theorem. If there is no transition of the signal *v* between *x+* and *x–* then, the STG becomes non–persistent (refer Definitions 1&2). Similarly, if there is no transition of *v* between *y+* and *y–* or between *z+* and *z–* then the transitions of *z* and *y* can be grouped together with no other signal transition between them. Thus, the resultant STG becomes non–persistent (refer Definitions 1&2). The signal *v* is the essential context signal to the signal *z* because the STG becomes non–persistent whenever the transitions of the signal *v* are removed. Thus, the essential context signal must be interleaved with the transitions of the signal x and with the transitions of the signal y or the signal z. A similar theorem was proven using lock graphs in [18]. It can be observed in Figure 8(b) that the signal v is not a context signal to the signal z because the trigger signals *x* and *y* are interleaved. ■

There can be many candidates for an essential context signal to a given non–input signal. According to theorem 8, the logic implementation should contain at least one of the many possible essential context signals. All the signals except the one which is selected for the logic implementation can be considered as non–essential context signals. Figure 8(c) shows an STG where the signal *z* has the signals *u* and *v* as the two possible essential context signals. The logic implementation must have either *u* and *v* as an input signal to realize the non–input signal *z*.

Figure 7(a) gives another example demonstrating the significance of the essential context signal. The signal *c* is the essential context signal of the non–input signal *b*. The trigger signals of *b* are the signals *a* and *d*. It can be observed that *a+* and *a–* transitions can occur between the transitions *b–* and *b+*. The essential context signal *c* is interleaved with both the trigger signals *a* and *d* as required by the theorem 8. Theorem 7 also says that the signal *c* is a context signal to the signal *b* because the transitions *c+* and *b+* are concurrent. In Figure 7(b), the signal *c* is a non–essential context signal of the signal *b*, because the transitions *a+* and *a–* no longer can occur between the transitions of *b*.



(i) STG

(ii) Contracted STG for signal y

(a) An Example with context signals

(i) STG

(ii) Contracted STG for signals z and y

(iii) Contracted STG for signal v

(b) An Example without context signals

(i) STG

(ii) Contracted STG for signal z – u is essential context signal of z

(iii) Contracted STG for signal z – v is essential context signal of z

(c) An Example with two contracted STGs for a signal

Figure 8. Examples illustrating the relationship between context signals and net contraction

6 Net Contraction

The behaviour of a non–input signal *s* can be represented by an STG which only contains transitions of *s* and $S_i(s)$ (input set of *s*) where $S_i(s)$ denotes the minimum set of the signals required to implement that signal *s*. This STG can be derived from the original one using a technique called net contraction [3]. A contracted STG for signal *s* is obtained from the STG by removing the transitions of the signals which are not in the set $s \cup S_i(s)$, in such a way that the concurrency and sequence relations between the remaining transitions are preserved.

Net contraction is done in the following way [3]. Let *t* be a transition of a signal that has to be eliminated from the STG. For all transitions t_i and t_j in the STG, add the arc $t_i \rightarrow t_j$ only if the arcs $t_i \rightarrow t$ and $t \rightarrow t_j$ are present. Then remove the transition *t* and all arcs connected to *t*.

Figure 8 illustrates the relation between the context signals and the net contraction. It is obvious that all the trigger signals of *s* are part of the set of input signals, $S_i(s)$ for the logic generating *s*. It was initially incorrectly believed that $S_i(s)$ only consists of trigger signals [3, 5]. But, the essential context signals of *s* should be part of $S_i(s)$ because without them the signal *s* becomes non–persistent [19]. The non–essential context signals of *s* can be removed from the STG to generate contracted STG for signal *s*. Thus, the input signals necessary to realize the logic for a signal *s* are the trigger signals and the essential context signals (if any) of *s* without which the *s* becomes non–persistent. The trigger signals can be easily detected in the STG. The essential context signals can be detected using Theorem 8. Thus we have provided the STG syntactic constraints which determine the set of input signals needed to generate the contracted STGs for all non–input signals.

In Figure 8(a) the signal *v* is the essential context signal of the signal *z* and the signal *x* is a non–essential context signal of the signal *y*. All signals except *v*, are trigger signals to the signal *v*. Thus only signal *y* has a contracted STG that is different from the original STG. The STG specification will become much simpler by just adding the constraint $x+ \rightarrow y-$ as shown in Figure 8(b). There are no context signals and all the non–input signals have the corresponding contracted STGs.

Chu [3] advocated the decomposition by net contraction as a means of obtaining efficient implementations. In the following theorem, we show that net contraction can be used to facilitate analysis and synthesis of the systems in an efficient manner but it does not produce any more efficient solutions.

Theorem 9 : *In a persistent STG, no signal except the trigger and the essential context signals of a non–input signal, effects the logic implementation of that non–input signal.*

Proof : ($\Rightarrow$) Let *z* be a non–input signal whose logic is being implemented. Let us consider a signal *x* which is neither a trigger nor an essential context signal of the signal *z*. The persistency of an STG ensures that the implied value of the signal *z* changes only in the states where its transitions are enabled. All states between *z+* and *z–* including the state in which *z+* is enabled have the same implied value for signal *z*. Similarly all states between *z–* and *z+* including the state in which *z–* is enabled have the same implied value for signal *z*. The signal *x* can be proved to be redundant in the logic implementation of the signal *z* if a pair of states with same signal values except for the signal *x* do not have different implied values for the signal z. That is, the states around a transition of the signal x should have the same implied value for the signal *z*. The theorem has to be proved for two different cases: (a) a transition of the signal *x* which is not concurrent with a transition of signal z or (b) a transition of the signal *x* which is concurrent with a transition of the signal z.

Let us assume that a transition of the signal *x* is not concurrent with a transition of the signal *z*. Let the values of *z* and *x* just before *x** be *Z* and *X*. Let # represent values all other signals. Thus, *#ZX* represent the signal values of a state in which the transition *x** is enabled and $\#Z\overline{X}$ represent the signal values of the resultant state due to the firing of the transition *x**. The implied value of *z* in both the states (i.e just before and just after transition *x** or *x*~*) is equal to *Z* because *x* is not a trigger signal. The implied value of *z* in state $\#Z\overline{X}$ is equal to $\overline{Z}$ if and only if *x* is a trigger signal of *z*. That means, as long as x is not trigger signal of z, all the adjacent vertices in the direction of signal *x* in the boolean n–cube have the same (including *don't–care*) implied value of *z*. Thus the signal *x* is redundant and does not affect the logic implementation.

Let us assume that a transition *x**, of the signal *x* is concurrent with a transition *z**, of the signal *z*. Since the transitions are concurrent, either *x** can occur before *z** or *z** can occur before *x**. Let the values of *x* and *z* before their transitions be *X* and *Z*. Let # represent values of all other signals. Thus, if *x** occurs before *z**, the two states just before and just after the transition *x** will have the binary code $\#ZX$ and $\#Z\overline{X}$ respectively. But, the implied value of the signal *z* in both the states is equal to $\overline{Z}$ because *z** can also occur before *x**. Similarly, if *z** occurs before *x**, the two states just before and just after the transition *x** will have the binary code $\#\overline{Z}X$ and $\#\overline{Z}\overline{X}$ respectively. The implied value of the signal *z* in both the states is equal to $\overline{Z}$. Thus it can be easily seen that all the adjacent vertices in the direction of the signal *x* in the boolean n–cube have the same (including *don't–care*) implied value of *z*. That means, the signal *x* is redundant and does not affect the logic implementation.

(⇐) Let us consider a signal *x* which is an essential context signal of the signal *z*. If the signal *x* is removed from the graph, the signal *z* will become non–persistent (see above for Definition 5). That means there will be two states with the same binary code and with two different transitions enabled (*z** being one of them). By introducing *x** between the two states, the states will have two different binary codes with the value of the signal *x* being different. The implied values of the signal *z* will be different in those two states. Thus, the signal *x* is not redundant and is necessary to implement the signal *z*. If *x* is a trigger signal to the signal *z*, it is obvious that *x* is necessary to implement the logic. ■

The above theorem proves that the trigger and the essential context signals of a non–input signal are necessary and sufficient to implement that non–input signal. It also proves that net contraction on a persistent STG does not give any more efficient solution. Figure 8(c) gives an example where the signal *z* has two possible essential context signals *u* and *v*. Thus, there are two contracted STGs for the signal *z* which contain either *u* or *v* as shown in Figure 8(c).

7 Hazard–free Logic Implementation

7.1 Logic derivation from a state graph

Assuming a given STG is live and satisfies CSC property, the state graph can be generated to derive the logic. The logic for each non–input signal can be determined from the implied values of the non–input signal in each state of the SG. For each non–input signal, a state in SG can be classified as either an on–set vertex or an off–set vertex in the n–dimensional boolean cube (*n* is the number of trigger and essential context signals). The vertices in the boolean cube that are not represented by the states in SG, form the don't–care set.

Once the on–set and off–set are derived from the SG, logic minimisation may be applied. The goal of minimisation is to find a hazard–free cover with a minimum number of cubes, while making each cube depend on as few literals as possible. Each cube in the minimum cover is a product term for a sum–of–product (SOP) implementation and a sum term for a product–of–sum (POS) implementation. The syntactic constraints defined in Section 3 only remove undesirable behaviour at functional level and do not produce automatically a hazard–free implementation. The following theorem proved in [13] gives the exact implications of the syntactic constraints.

Theorem 10 : *If the given STG is live and satisfies CSC property, then every non–input signal is a unate function of itself.*

Proof : Let us assume for the sake of contradiction that an output variable x_i is not unate in itself. Then there must be an on–set vertex $v' = (x_1, x_2, \ldots, x_i = 0, \ldots, x_n)$ and an off–set vertex $v'' = (x_1, x_2, \ldots, x_i = 1, \ldots, x_n)$. The vertices v' and v'' indicate that the variable x_i is controlling itself. That means there is an inherent oscillation due to the negative feedback, which violates the CSC assumption. ■

The above theorem implies that every output variable t can be expressed as $t = S + Mt$ (feedback loop SOP realisation) where S and M are combinational logic functions independent of t. Here, S which initially sets the value of t to 1, is called the *s–cover* and Mt which maintains the value of t until the transition t– occurs, is called the *m–cover*. Thus, the following latches may be used to implement the next–state logic [13] :

S–M Latch : $t = S + M\,t$ [1]

S–R Latch : $t = S + \overline{R}\,t$ where $M = \overline{R}$

C–element : $t = AB + (A{+}B)\,t$ where $S = AB$ and $M = (A{+}B)$

If POS implementation is required, the output variable t can be expressed as $t = R\,.\,(M' + t)$ (feedback loop POS realisation) where R, which resets the value of t, is called the *r–cover* and $(M'{+}t)$ which maintains the value of t as 0 until the transition t+ occurs, is called the *m'–cover*.

In Section 6, we have shown that the trigger and the essential context signals of f_i are necessary and sufficient to realise a logic function f_i. Thus the s–cover, m–cover, r–cover and m'–cover are functions of the trigger and the essential context signals of f_i.

7.2 Hazard–free logic implementation

In this Section, we analyze logic hazards under both the single input change (SIC) and the multiple input change (MIC) conditions, and give the synthesis procedures needed to remove logic hazards under both conditions of operation, assuming the unbounded–gate delay model. The required changes to the net contraction procedure and existing logic synthesis methods in realizing hazard–free implementation are also discussed.

7.2.1 Hazard–free implementation under SIC condition

A two–level implementation has only one type of logic hazard, namely the 1–static hazard in a SOP implementation and 0–static hazard in a POS implementation [16].

The circuit C_i is free of static hazards as long as at least one of its cubes remains constant during each transition of its input signals. As long as one cube output is constant, the output of the circuit C_i remains constant (1 in SOP implementation and 0 in POS implementation) and thus the circuit is hazard–free. This is achieved by covering every two adjacent states in SG or every signal transition in STG with the same cube. A simple algorithm which removes all hazards under SIC conditions is given in [13] for SOP implementation. A similar algorithm can be obtained for POS implementation. There are no changes to the net contraction procedure to remove hazards under SIC conditions.

7.2.2 Hazard–free implementation under MIC condition

Concurrent signal transitions which occur simultaneously can also cause hazards. We must identify the cubes (product terms in SOP and sum terms in POS) which can potentially produce glitches under MIC conditions. Under the unbounded–gate delay and zero–wire delay model, multiple input changes correspond to concurrent signal transitions in a contracted STG. Thus the effect of concurrent signal transitions on cubes should be analyzed for the following four output transitions : $0 \Rightarrow 0$ and $1 \Rightarrow 1$ (for static hazards), $1 \Rightarrow 0$ and $0 \Rightarrow 1$ (for dynamic hazards).

In [13], a procedure to remove all the hazards under MIC conditions is given. Here, the algorithm for minimizing logic under SIC condition is used first to obtain the *SOP realisation*. Then all the cubes which can produce glitches during the output transitions : $0 \Rightarrow 0$, $1 \Rightarrow 0$ and $0 \Rightarrow 1$, are identified. Once the problematic cubes are identified, the glitches are eliminated by adding to each such cube some literal which remains at 0 while concurrent transitions take place. However, if no literal which remains at 0 is available, then the circuit is not hazard–free under some concurrent transitions. After such transitions are identified, the STG can be modified to sequentialize the concurrency in such a way that no MIC hazards exist [13].

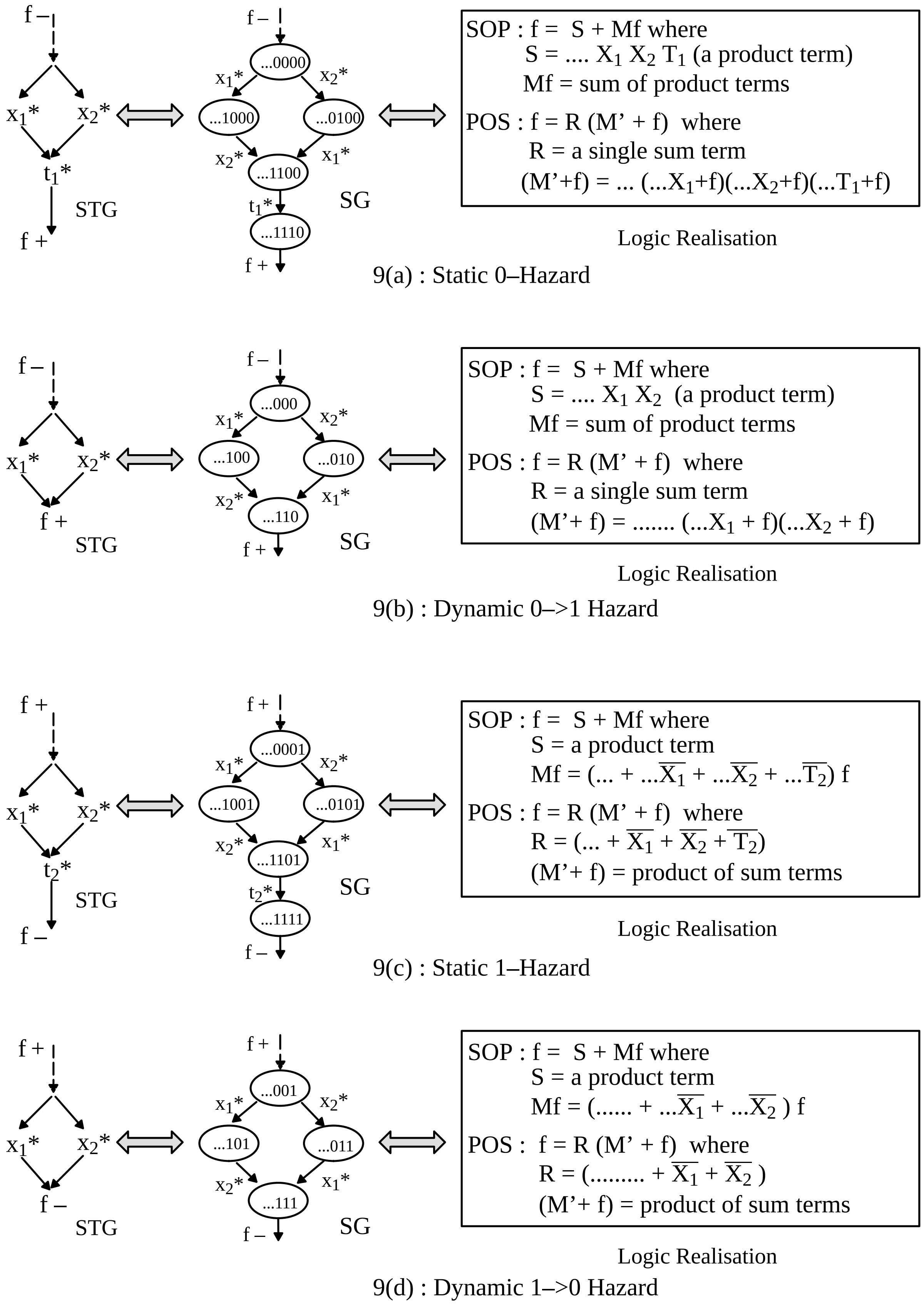
In this Section, we show that if the *feedback loop SOP (POS) realisation* is hazard–free under SIC condition, it is always hazard–free under MIC condition during all the output transitions except the $1 \Rightarrow 0$ ($0 \Rightarrow 1$) transition. We give the STG conditions for which the SOP realisation is not hazard–free and provide the necessary changes to the net contraction and to the logic synthesis algorithms, so that the synthesized circuit is hazard–free (under both SIC and MIC conditions).

Let us consider n concurrent signal transitions $x_1*, x_2*, \ldots, x_n*$ where $x_1, x_2, \ldots, x_n$ are either trigger or essential context signals of the output signal f. Let the values of $x_1, x_2, \ldots, x_n$ before firing the transitions be $\overline{X_1}, \overline{X_2}, \ldots, \overline{X_n}$ and after firing the transitions be $X_1, X_2, \ldots, X_n$. Let t_1* (t_2*) represent all transitions between x_i* and $f+$ ($f–$) excluding $x_i*\sim$, where the concurrent signal transitions x_i* are not the trigger transitions of $f+$ ($f–$). Let the values of t_1 and t_2 before firing the transitions be $\overline{T_1}$ and $\overline{T_2}$ and after firing the transitions be T_1 and T_2. The interrelationship between the concurrent transitions in the STG and the possible hazards associated with the resultant logic realisation is discussed below. For convenience sake, let us consider n=2.

- For the output f to remain 0 (f : $0 \Rightarrow 0$) during the concurrent transitions , the transitions x_1* and x_2* should occur between the transitions $f–$ and $f+$, and x_1* and x_2* should not be the trigger transitions of f–. The corresponding STG and the resultant cubes (for SOP and POS) are given in Figure 9(a).
- Similarly, for the output f to remain 1 (f : $1 \Rightarrow 1$) during the concurrent transitions, the transitions x_1* and x_2* should occur between the transitions $f+$ and $f–$, and x_1* and x_2* should not be the trigger transitions of $f+$. The corresponding STG and the resultant cubes (for SOP and POS) are given in Figure 9(c)
- The concurrent transitions x_1* and x_2* cause the output transition $f+$ (f : $0 \Rightarrow 1$) if and only if x_1* and x_2* are the trigger transitions of $f+$. The corresponding STG and the resultant cubes (for SOP and POS) are shown in Figure 9(b).
- Similarly the concurrent transitions x_1* and x_2* cause the output transition $f–$ (f : $1 \Rightarrow 0$) if and only if x_1* and x_2* are the trigger transitions of $f–$. The corresponding STG and the resultant cubes (for SOP and POS) are shown in Figure 9(d).

The illustrations given in Figure 9 can be generalized for n concurrent signal transitions. The logic realisations deduced in Figure 9 by no means represent all possible STGs, but can be used to analyze for MIC hazards. In particular, the STGs in which both a transition and its complementary transition occur between f+ (f–) and f– (f+), are not considered in Figure 9. Let us consider $f– \xrightarrow{*} p* \xrightarrow{*} p*\sim \xrightarrow{*} f+$. Let P ($\overline{P}$) be the value of p between $p*$ ($p*\sim$) and $p*\sim$ ($p*$). It can be observed easily that the s–cover in SOP contains $\overline{P}$ as a literal and m'–cover in POS contains ($\overline{P}$+f) as a cube. Similarly if we consider $f+ \xrightarrow{*} p* \xrightarrow{*} p*\sim \xrightarrow{*} f–$, the m–cover in SOP contains Pf as a cube and the r–cover in POS contains P as a literal.

The illustrations given in Figure 9 support single occurrences of the +ve and –ve transitions ($f+$ and $f–$) of an output signal f. If multiple +ve and –ve transitions of an output signal f are allowed in an STG, then the logic realisations given in Figure 9 are still valid except that the



SOP : f = S + Mf where
S = ... X_1 X_2 T_1 (a product term)
Mf = sum of product terms

POS : f = R (M’ + f) where
R = a single sum term
(M’+f) = ... (...X_1+f)(...X_2+f)(...T_1+f)

Logic Realisation

9(a) : Static 0–Hazard

SOP : f = S + Mf where
S = ... X_1 X_2 (a product term)
Mf = sum of product terms

POS : f = R (M’ + f) where
R = a single sum term
(M’+ f) = ... (...X_1 + f)(...X_2 + f)

Logic Realisation

9(b) : Dynamic 0–>1 Hazard

SOP : f = S + Mf where
S = a product term
Mf = (... + ...$\overline{X_1}$ + ...$\overline{X_2}$ + ...$\overline{T_2}$) f

POS : f = R (M’ + f) where
R = (... + $\overline{X_1}$ + $\overline{X_2}$ + $\overline{T_2}$)
(M’+ f) = product of sum terms

Logic Realisation

9(c) : Static 1–Hazard

SOP : f = S + Mf where
S = a product term
Mf = (...... + ...$\overline{X_1}$ + ...$\overline{X_2}$) f

POS : f = R (M’ + f) where
R = (......... + $\overline{X_1}$ + $\overline{X_2}$)
(M’+ f) = product of sum terms

Logic Realisation

9(d) : Dynamic 1–>0 Hazard

Figure 9. Illustration of the interrelationship between concurrent signal transitions in an STG ($x_1* = x_1+$, $x_2* = x_2+$, $t_1* = t_1+$, $t_2* = t_2+$) and the possible hazards associated with the resultant logic realisation

s–cover, *S* (r–cover, *R*) may contain multiple product terms (sum terms) and m–cover, *Mf* (M'–cover, *(M'+f)*) may contain more product terms (sum terms). For example, if the STG has two f+ and two f– transitions then s–cover may be sum of two product terms (p–terms) and r–cover may be product of two sum terms (s–terms). The feedback loop SOP and POS logic realisations are now analysed for hazards under MIC conditions.

Lemma 1 : The feedback loop SOP (POS) realisation of *f* is free of static 1–hazards (static 0–hazards) under MIC condition if the realisation is hazard–free under SIC condition [16,13].

Proof : Let us consider concurrent transitions associated with static 1–hazards, that is the concurrent transitions are not the trigger transitions of f– and occur between the transitions *f*+ and *f*– (see Figure 9(c)). The procedure used to remove hazards under SIC conditions guarantees the existence of a constant 1 cube during the concurrent transitions [16, 13]. Thus the SOP realisation is free of static 1–hazards under MIC condition. ■

Theorem 11 : The feedback loop SOP (POS) realisation of *f* is free of static 0–hazards (static 1–hazards) under MIC condition, if the realisation is hazard–free under SIC condition.

Proof : The feedback loop SOP realisation is free of static 0–hazards if the combinational functions *s–cover* and *m–cover* are free of static 0–hazards under MIC conditions. The concurrent transitions with which the static 0–hazards are associated, are not the trigger transitions of *f*+ and occur between the transitions *f*– and *f*+. From Figure 9(a), we can see that the *s–cover* is a product term (p–term) and *m–cover* is sum of a number of product terms. The *m–cover* does not produce any glitch because all the product terms contain *f* as a literal with the value of *f* being 0. The p–term of *s–cover* may produce a glitch if it has no literal equal to 0 during the concurrent transitions and if it has complementary appearances of literals corresponding to the concurrent transitions. During the concurrent signal transitions the values of all other signals remain constant. The p–term of the *s–cover* will contain a literal which is equal to 0 until the transition t_1*, if only the transition t_1* (and not also t_1*~) occur between the concurrent transitions and *f*+. If $t_1* = t_1+$ then the s–cover will contain T_1 as a literal and if $t_1* = t_1-$ then the s–cover will contain $\overline{T_1}$ as a literal. Thus even if the *s–cover* contain any combination of literals, the *s–cover* will be zero during the concurrent transitions. A transition such as t_1* always exists if the given STG has the CSC property.

Let us now consider an STG with two *f*+ and two *f*– transitions. Let P_1 and P_2 be the two product terms in the *s–cover* of the feedback loop SOP realisation. The p–terms, P_1 and P_2 represent the two states S_1 and S_2 of the state graph in which two different *f*+ (lets say f_1+, f_2+ respectively) transitions are enabled. The p–term P_1 (P_2) is determined by the values of all the signals except *f*, in the state S_1 (S_2). From the first part of the proof, it is clear that P_1 and P_2 cannot produce any glitches between $f_1–$ and f_1+ and between $f_2–$ and f_2+ respectively. If the p–term P_1 (P_2) produces a logic–1 between $f_2–$ ($f_1–$) and f_2+ (f_1+) then the feedback loop SOP realisation is not free of static 0–hazards. The value of the p–term P_1 (P_2) can become 1 only in a state whose binary vector is same as the state S_1 (S_2), except the value of the signal *f*. Thus in order to generate logic–1 between $f_2–$ ($f_1–$) and f_2+ (f_1+), there should be a state in SG, between $f_2–$ ($f_1–$) and f_2+ (f_1+) which has the same binary vector as S_1 (S_2). That is, there should be two states with the same binary code in which a non–put signal transition f_1+ (f_2+) is enabled, which violates the CSC assumption. Thus the feedback loop SOP realisation is free of static 0–hazards. Similarly we can prove even if the given STG has multiple (more than two) occurrences of $f+$ and $f–$ transitions, that the feedback loop SOP realisation is free of static 0–hazards under MIC conditions.

Similarly, it can be shown that the feedback loop POS realisation is always free of static 0–hazards. ■

Theorem 12 : The feedback loop SOP (POS) realisation of *f* is free of the dynamic hazards associated with the 0 → 1 (1 → 0) output transition under MIC conditions, if the realisation is hazard–free under SIC conditions.

Proof : The feedback loop SOP realisation, $f = S + Mf$ is free of dynamic 0 → 1 hazards if the cubes in the s–cover and the m–cover do not produce glitches during the 0 → 1 transition of the output.

The concurrent transitions associated with the 0 → 1 output transition are the trigger transitions of $f+$. In the feedback loop SOP realisation, *s–cover* is a single p–term and *m–cover* is the sum of product terms as shown in Figure 9(b). The *m–cover* does not cause a glitch because all the p–terms in the *m–cover* contain *f* as a literal and the value of f is zero between f– and f+. The p–term of *s–cover* changes its value from 0 to 1 only when all the concurrent transitions occur (simultaneously or in any order). Thus the feedback loop SOP realisation is always free of the dynamic hazards associated with the 0 → 1 output transition.

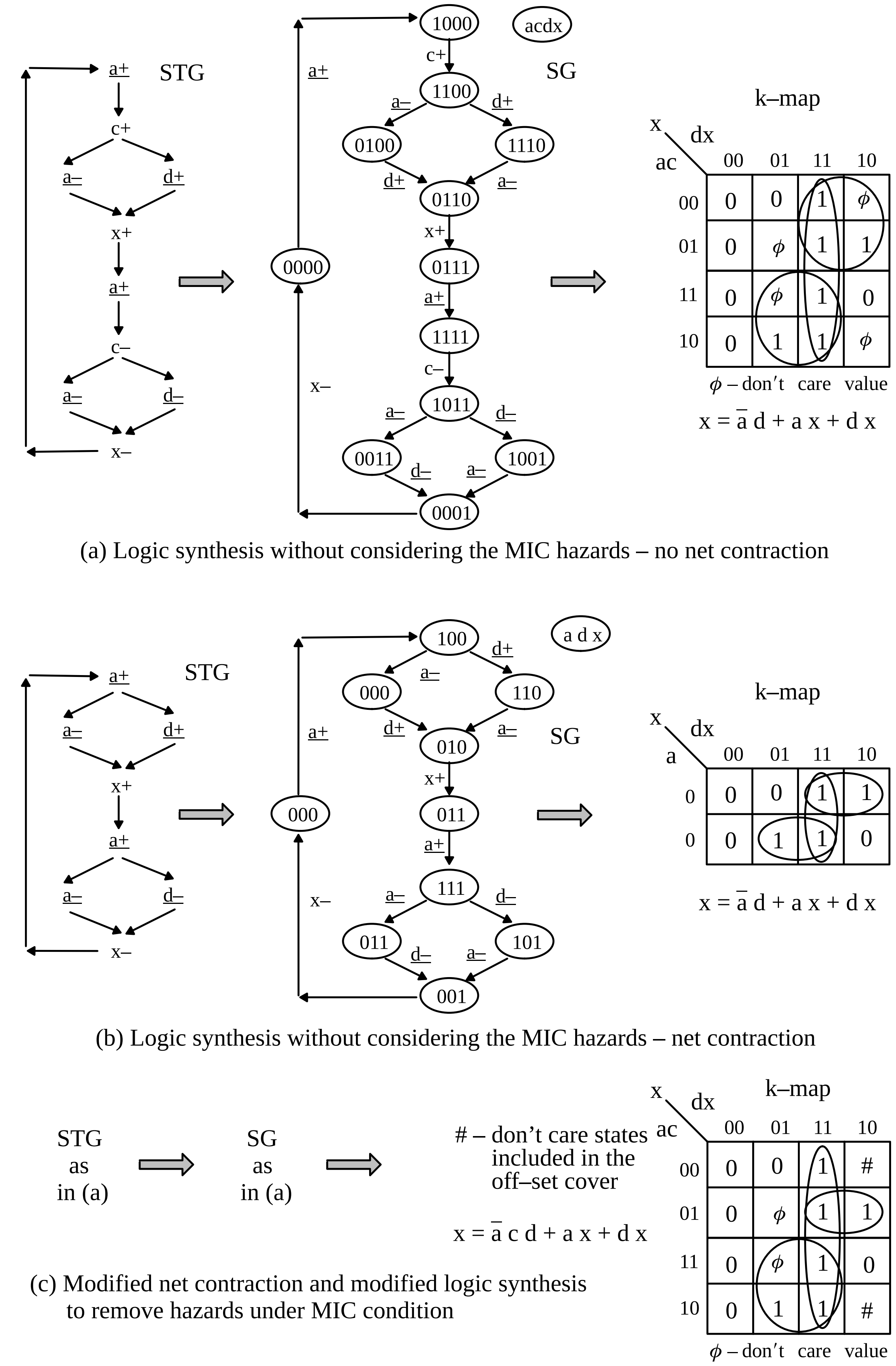
Let us now consider a case where multiple +ve and –ve output signal transitions ($f+$ and $f–$) are allowed in an STG. The s–cover consists of multiple p–terms, P_1, P_2, P_3, ..., P_n representing the states, S_1, S_2, S_3, ..., S_n of the state graph in which different $f+$ transitions, f_1+, f_2+, f_3+, ..., f_n+ are enabled. The value of the p–terms P_1, P_2, P_3, ..., P_n can become 1 only in the states whose binary vectors are same as that of the states S_1, S_2, S_3, ..., S_n, ignoring the value of *f*. Thus in order to generate a glitch, there should be a new state between $f_n–$ and f_n+ whose binary vector is same as that of S_1, S_2, S_3, ..., S_n which violates CSC assumption. Thus even if the given STG has multiple occurrences of $f+$ and $f–$ transitions, the feedback loop SOP realisation is free of the dynamic hazards associated with the 0 → 1 output transition.

Similarly, we can prove that the feedback loop POS realisation is always free of the hazards associated with the 1 → 0 output transition. ■

The following Theorem establishes the STG conditions for which the hazards associated with the 1 → 0 (0 → 1) output transition occur under MIC condition in the feedback loop SOP (POS) realisation.

Theorem 13 : In the net contracted STG, let us consider the concurrent trigger transitions of $f–$ ($f+$) : $x_1*, x_2*, \ldots, x_n*$. Let $S_{on} = b_1...b_n\overline{F}$ represents the binary vector of the state in SG, in which the transition f+ is enabled, where $b_1...b_n$ denote the values of the trigger and essential context signals of *f*, in S_{on}. The SOP (POS) realisation is not free of the dynamic 1 → 0 (0 → 1) hazards if and only if a state associated with the concurrent transitions, x_i* in SG has the same binary vector as that of S_{on} except the value of f, i.e. $b_1...b_nF$.

Let y_i* represent all single signal transitions between the transition $f+$ ($f–$) and concurrent transitions, in the uncontracted STG. The SOP(POS) realisation consists of the dynamic hazards associated with the 1 → 0 (0 → 1) output transition if and



ac \ dx	00	01	11	10
00	0	0	1	ϕ
01	0	ϕ	1	1
11	0	ϕ	1	0
10	0	1	1	ϕ

$x = \overline{a}\, d + a\, x + d\, x$

(a) Logic synthesis without considering the MIC hazards – no net contraction

a \ dx	00	01	11	10
0	0	0	1	1
0	0	1	1	0

$x = \overline{a}\, d + a\, x + d\, x$

(b) Logic synthesis without considering the MIC hazards – net contraction

ac \ dx	00	01	11	10
00	0	0	1	#
01	0	ϕ	1	1
11	0	ϕ	1	0
10	0	1	1	#

$x = \overline{a}\, c\, d + a\, x + d\, x$

(c) Modified net contraction and modified logic synthesis to remove hazards under MIC condition

Figure 10. Logic synthesis and net contraction associated with MIC hazards

only if

(a) a complementary transition *of* x_i*, $x_i*\sim$ also occurs between $f+$ ($f–$) and x_i*,

(b) a complementary transition of x_j*, $x_j*\sim$ do not occur between $f+$ ($f–$) and x_i*,

(c) a transition of y_i* is not an essential context signal of f and

(d) a complementary transition of any transition of y_i*, $y_i*\sim$, is not a trigger transition of $f+$ ($f–$).

Proof : Figure 9(d) illustrates the concurrent transitions associated with the dynamic 1 → 0 hazard. The feedback loop SOP realisation, $f = S + Mf$ is not free of dynamic 1 → 0 hazards if the cubes in the *s–cover* and the *m–cover* produce glitches during the 1 → 0 transition of the output. The *m–cover* changes its value from 1 to 0 only when all the concurrent transitions occur (simultaneously or in any order). Thus the *m–cover* does not cause any glitch under MIC condition.

The dynamic 1 → 0 hazard can only occur if the *s–cover* causes a 0 → 1 → 0 glitch during the concurrent transitions and if the glitch propagates to the output f only after the output f becomes 0 due to the 1 → 0 change in the *m–cover*. The p–term of *s–cover* is equal to the product of the literals corresponding to the values of the trigger and essential context signals of f in the state S_{on}, i.e. $b_1...b_n$. The value of the p–term is equal to 1 in all the states whose binary vector is equal to either $b_1...b_n\overline{F}$ or $b_1...b_nF$. Thus in order to generate a logic–1 when *m–cover* changes its value from 1 to 0, a state associated with the concurrent transitions must have the binary vector equal to $b_1...b_nF$. The four conditions given above ensure that a state associated with the concurrent transitions always has the binary vector $b_1...b_nF$. Conditions (c) and (d) ensure that the transitions such as y_i* will be removed in the net contracted STG and the prime and irredundant *s–cover* is independent of the literals corresponding to the transitions y_i*. The Condition (a) implies that the signal x_i must have more than two transitions and a same transition x_i* also occurs between $f–$ and $f+$. Condition (a) also implies that the s–cover can cause 0 → 1 transition of f, during the concurrent transitions. Condition (b) ensure that the *s–cover* always stabilises back to 0 in accordance with the $f–$ transition. Thus, the p–term of the *s–cover* causes the dynamic hazard. If the transition $y_i*\sim$ is a trigger transition of $f+$ (Condition (d) not satisfied) or if the transition y_i* is an essential context signal of f (Condition (c) not satisfied), then the p–term contains $\overline{Y_i}$ (the value y_i after $y_i*\sim$) as a literal. When the transition y_i* occurs, the p–term of the *s–cover* becomes 0 and does not produce the glitch.

Similarly we can prove that the POS realisation is not free of the hazards associated with 0 → 1 output transition. ■

In an un–contracted STG, if suppose $y_i*\sim$ is not the trigger transition of $f+$, but occurs between $f–$ and $f+$, then the *prime s–cover* (with don't cares) does not contain $\overline{Y_i}$ as a literal, but the *non–prime s–cover* (with out don't cares) contains $\overline{Y_i}$ as a literal. This property is used to remove the hazards. The contraction procedure normally removes transitions such as y_i* and $y_i*\sim$. Figure 10(a) & 10(b) give an implementation of the signal x which is not hazard–free under MIC conditions. Figure 10(a) illustrates the synthesis procedure without the net contraction where as Figure 10(b) illustrates synthesis with net contraction.

In order to eliminate MIC hazards, the net contraction procedure should not remove at least one set of transitions, such as y_i* (between $f+$ and $f–$) and $y_i*\sim$ (between $f–$ and $f+$). The next step in the synthesis process is to add the don't care states in which Y_i (the value after y_i*) appears, to the off–set cover. This process will add the literal $\overline{Y_i}$ to the p–term of *s–cover*. Figure 10(c) gives an implementation of the signal *x* which is hazard–free under MIC condition and also illustrates the modifications in the net contraction and the logic synthesis. In this example the net contraction procedure finds *c*– and *c*+ as y_i* and y_i*~ transitions and thus the STG and the state graph remain the same as shown in Figure 10(c). During logic synthesis, the don't care states *0010* and *1010* in which the literal *c* is equal to 0, are included in the off–set cover.

If we cannot find transitions such as y_i* and y_i*~ then the realisation is not hazard–free under the concurrent occurrences of some transitions. In the following Theorem we show that the transitions such as y_i* (between *f*+ and *f*–) and y_i*~ (between *f*– and *f*+) can always be found if the STG is live, safe and satisfies the CSC property.

Theorem 14 : If the given STG is live, safe and satisfies the CSC property, then a hazard–free feedback loop SOP (POS) implementation (under both SIC and MIC conditions) always exists unless the concurrent trigger transitions of the output transition *f*–, are associated with the input signals which come from the same environment. If the environment from which the input signals come, also provide an extra signal to implement the feedback loop SOP (POS) realisation, then the implementation can be made always hazard–free.

Proof : The procedure to remove all the hazards under SIC condition is explained in Section 7.2.1. If the implementation is hazard–free under SIC conditions, then the only hazards that can occur, are associated with the output transition 1 → 0 and with the MIC conditions (from Lemma 1 and Theorems 11–13). That is, the implementation is not hazard–free if and only if the first two STG conditions given in Theorem 13 are satisfied. Those conditions are reflected in the STG shown in Figure 10(b) where the *x*– is the 1 → 0 output transition and *a*– and *d*– are the concurrent signal transitions associated with MIC condition. If we can prove that there will be always signal transitions such as *c*+ and *c*– as shown in Figure 10(a), then the hazards associated with the 1 → 0 output transition can be removed by adding a literal of *c* to the problematic p–terms of the s–cover.

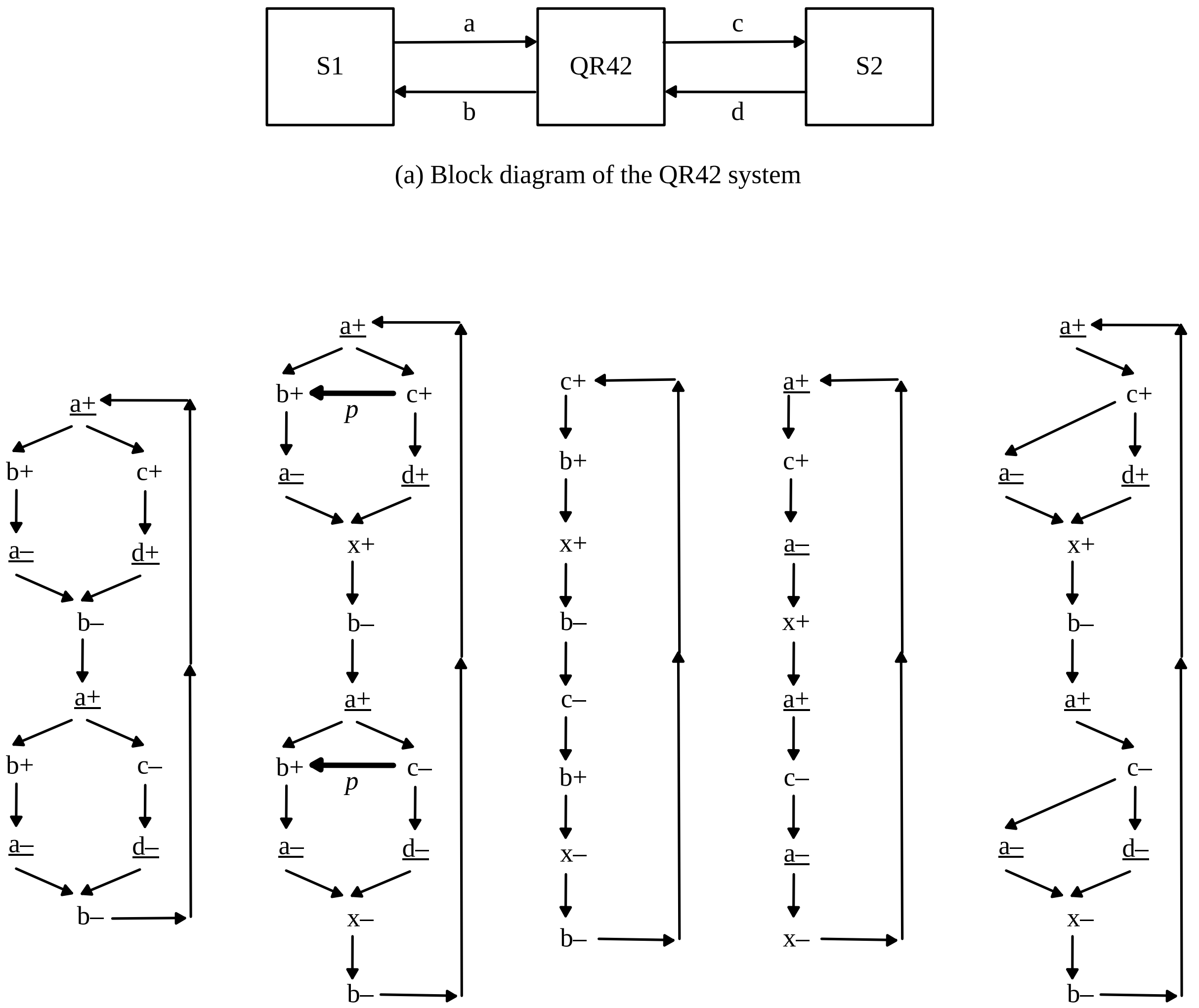
If either *a* or *d* is not an input signal, then the STG in Figure 10(b) does not satisfy the CSC property. In order to enforce CSC, without removing the concurrency between *a*– and *d*–, a signal transition such as *c*– should be there.

If *a* and *d* are input signals but come from two independent environments then we can not have the causal constraint $\underline{a+} \rightarrow \underline{d-}$ without having at least one signal transition such as *c*– in between them.

If *a* and *d* are input signals and come from same environment then they can be considered as output signals to that environment. There should be at least a signal transition such as *c*– in that environment. Otherwise there will be CSC violation in that environment. If the signal such as c is available from that environment then the circuit is hazard–free under concurrent occurrences of the input signal transitions. Otherwise, the circuit is not hazard–free for the given STG specification.

In all the three cases, the inverted transition of *c*– should always occur between *x*– and *x*+. Otherwise there will be a CSC violation.

Thus if the STG is live, safe and satisfies CSC then there will be always a feedback loop SOP implementation which is hazard–free under both SIC and MIC conditions. Similarly we can prove that a hazard–free feedback loop POS implementation always exists. ■



(a) Block diagram of the QR42 system

(b) Original STG (c) Final STG (d) Contracted STG for *b* (e) Contracted STG for *c* (f) Contracted STG for *x*

Figure 11. Synthesis of QR42 module

8 Examples

In this Section two examples will be discussed to demonstrate the significance of the theory developed above. The different aspects of the synthesis process such as persistency or complete state coding, trigger and context signals, net contraction and hazard removal are described.

8.1 Module QR42

Module QR42 [5] can be used to connect two systems S1 and S2 as shown in Figure 11(a), where S1 follows four–phase signaling and S2 follows two–phase signaling. Assuming that all the signals are at low voltage initially, one cycle of QR42 activities can be specified as

$$[\underline{a+}; ((b+; \underline{a-}) \parallel (c+; \underline{d+})); b-; \underline{a+}; ((b+; \underline{a-}) \parallel (c-; \underline{d-})); b-]^*$$

A sequence $b+$; $\underline{a-}$ indicates that $\underline{a-}$ should come after $b+$. The notation $b+ \parallel c+$ denotes that $b+$ and $c+$ can occur in parallel. The notation $[a]^*$ indicates zero or more repetitions of a. The STG that captures the signaling specifications of QR42 is given in Figure 11(b). The initial marking of this STG consists of the edge leading into the topmost $a+$ transition. The STG has two occurrences of the same transitions of the signals a and b. There is no other signal transition between two occurrences of $s^* = \{b+, a–\}$ except $s^{*\sim} = \{b–, a+\}$. This clearly violates the Restriction 1, which allows multiple occurrences of the same transitions as long as they are distinguished by a signal transition in between them. An internal signal transition cannot be inserted before an input signal transition (e.g. between $b–$ and $a+$) because the circuit can not impose constraints on the outside world. Thus the internal signal transitions $x+$ and $x–$ are inserted as shown in Figure 11(c). The STG is still non–persistent due to the constraints $a+ \rightarrow c+$ and $a+ \rightarrow c–$ and the two occurrences of the sequence of transitions $\{b–, a+, b+, a–\}$. The persistency constraints $c+ \rightarrow b+$ and $c– \rightarrow b+$ are introduced as shown in Figure 11(c) to make the STG persistent. The logic equations resulting from the boolean minimisation of the final STG are given below.

$$b = \overline{c}x + c\overline{x}$$

$$c = a\overline{x} + c(\overline{a} + \overline{x})$$

$$x = \overline{a}d + x(a + d)$$

before removing hazards under MIC condition

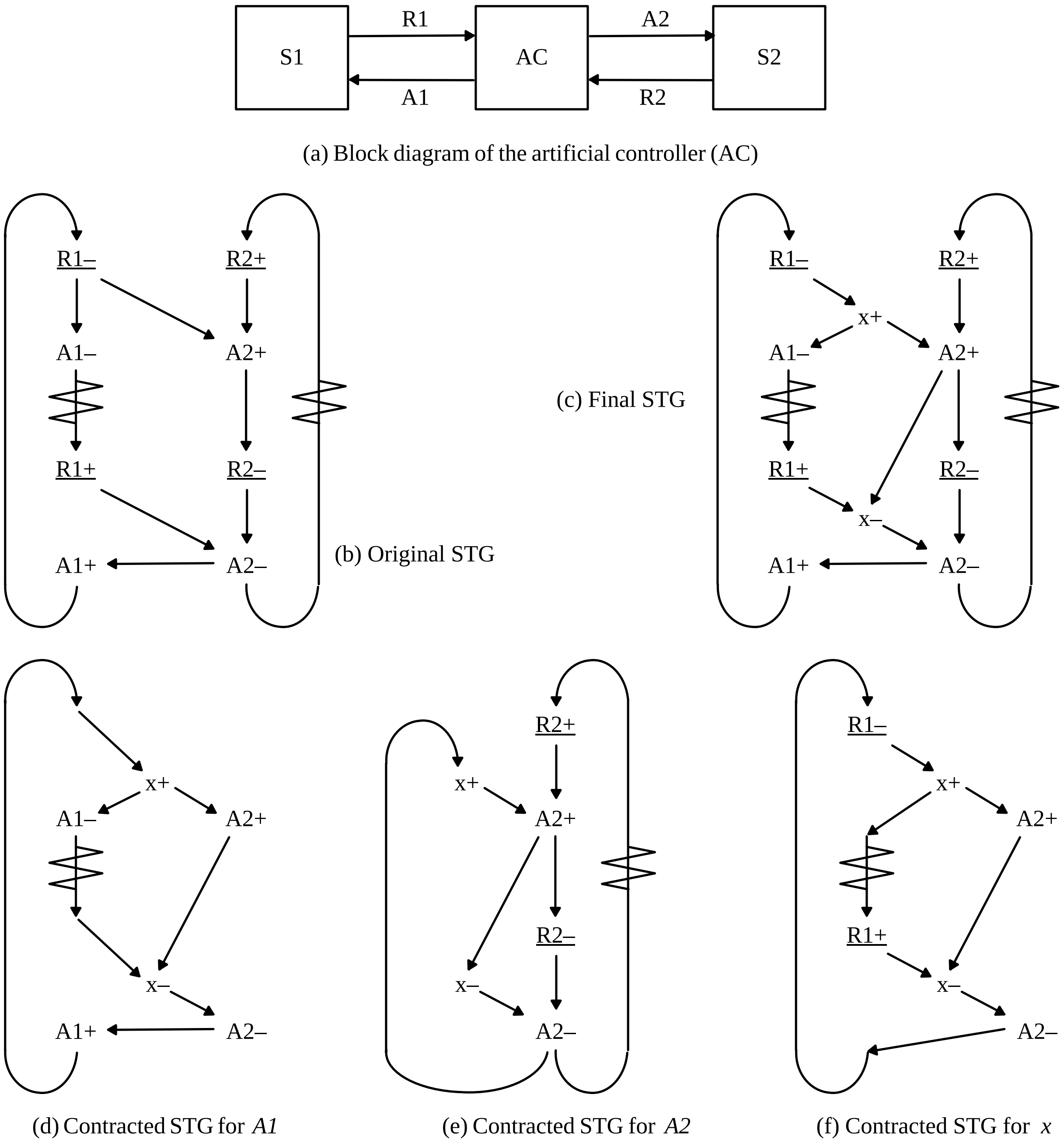
$$x = \overline{a}cd + x(a + d)$$

after removing hazards under MIC condition

The same logic equations are also obtained from the contracted STGs. The trigger signals for the signal b are c and x because of the arcs $c+ \rightarrow b+$ and $x+ \rightarrow b–$. There is no essential context signal for the signal b. The resultant contracted STG for the signal b is shown in Figure 11(d). In the case of signal c, the trigger signal is a and the essential context signal is x. If x is removed the signal c becomes non–persistent. The contracted STG for the signal c is given in Figure 11(e). The trigger signals for the signal x are a and d. In order to remove the hazards under MIC condition, the transitions of the signal c should be included in the contracted STG. The resultant contracted STG is shown in Figure 11(f). If the don't–care states associated with c=0 (c– transition) are not included in the off–set, then the resultant logic still contain the hazards under MIC condition.

8.2 An Artificial Controller

In this Section, an artificial controller given in [18] will be discussed. The artificial controller can be used to connect two systems S1 and S2 which follows four–phase signalling as shown in Figure 12(a). The corresponding STG is given in Figure 12(b). The important constraint here is 5that the time critical transitions of the two systems (i.e. $\underline{R_1+}$ and $\underline{R_2+}$) should not occur on a simple cycle. The arcs $R_1– \rightarrow A_2+$ and $R_1+ \rightarrow A_2–$ are non–persistent with respect to A_1+. The constraint arc $A_2+ \rightarrow A_1–$ can be used to remove the non–persistency behaviour. In that case, the two time consuming transitions reside on a simple cycle, which is not allowed. An internal signal transition x+ can be introduced as shown in Figure 12(c) so that the transitions of A_2 are persistent with respect to A_1 while the two time consuming transitions are not on a simple cycle. It can also be observed that the transition A_1+ is not persistent with respect to R_2+. Thus, the transition x– is inserted between A_2+ and $A_2–$ as shown in Figure 12(c) in such a way that the transition A_1+ is persistent and the transitions x+ and x– are on a simple cycle. There are no essential context signals for all the non–input signals in the final STG. The contracted STGs for the non–input signals A_1, A_2 and x are given in Figures 12(d), 12(e) and 12(f) respectively. The following logic equations can be obtained either directly from the boolean minimisation of the final STG or from the contracted STGs.



(a) Block diagram of the artificial controller (AC)

(b) Original STG

(c) Final STG

(d) Contracted STG for *A1*

(e) Contracted STG for *A2*

(f) Contracted STG for *x*

Figure 12. Synthesis of the artificial controller

$$A_1 = \overline{A_2}\ \overline{x}$$

$$A_2 = R_2\ x + A_2(R_2 + x)$$

$$x = \overline{R_1} + \overline{A_2}\ x$$

9 Conclusions

In this paper, we dealt with signal transition graph constraints for hazard–free synthesis of asynchronous circuits with unbounded–gate delay model. We established a relationship between Chu’s persistency constraint [3] and the complete state coding constraint [7, 13] with the help of global persistency constraints. We provided the STG syntactic constraints required to identify the *input set* of a signal. We showed contrary to Chu's belief [3] that the net contraction does not produce any solution more efficient than the solution obtained using boolean minimisation from an un–contracted STG. We analyzed hazards under both single and multiple input change conditions and proposed the necessary changes to the net contraction and the logic synthesis procedures in order to obtain hazard–free implementations.

Acknowledgement

The authors would like to thank Ruchir Puri for providing us with his unpublished manuscript and for many useful discussions.

References

[1] C. Berthet and E. Cerny. Synthesis of Speed–independent Circuits Using Set–Memory Elements. In G. Saucier, editor, *Proc. International Workshop on Logic and Arch. Synthesis for Silicon Compilers*. Grenoble, France, May 1988.

[2] T.–A. Chu. Synthesis of Self–timed Control Circuits from Graphs: An Example. In *International Conference on Computer Design*, pages 565–571, October 1986.

[3] T.–A. Chu. *Synthesis of Self–timed VLSI Circuits from Graph–theoretic Specifications.* PhD thesis, MIT, June 1987.

[4] F. Commoner, A. W. Holt, S. Even and A.Pnueli. Marked Directed Graphs. *Journal of Computer and System Sciences*, 5:511–523, 1971.

[5] G. Gopalakrishnan and P. Jain. Some Recent Asynchronous System Design Methodologies. Technical Report UU–CS–TR–90–016, University of Utah, October 1990.

[6] M. Hack. Analysis of Production Schemata by Petri Nets. Master's thesis, MIT, 1972. (Project MAC TR–94).

[7] L. Lavagno, K. Keutzer and A. Sangiovanni–Vincentelli. Synthesis of Verifiably Hazard–free Asynchronous Control Circuits. In *Advanced Research in VLSI: UC Santa Cruz*, Pages 86–102, 1991.

[8] L. Lavagno, K. Keutzer and A. Sangiovanni–Vincentelli. Algorithms for synthesis of hazard–free asynchronous circuits. In *Proceedings of the Design Automation Conference*, Pages 302–308, June 1991.

[9] L. Lavagno, C. W. Moon, R. K. Brayton and A. Sangiovanni–Vincentelli. Solving the state assignment problem for signal transition graphs. In *Proceedings of the Design Automation Conference*, Pages 568–572, June 1992.

[10] T. H.–Y. Meng, R. W. Brodersen and D. G. Messerschnitt. Automatic Synthesis of Asynchronous Circuits from High–Level Specifications. *IEEE Transactions on CAD of Integrated Circuits,* 8(11):1185–1205, November 1989.

[11] T. H.–Y. Meng. *Synchronization Design of Digital Systems*. Kluwer Academic, 1990

[12] R. E. Miller. *Switching Theory*, volume II, chapter 10. John Wiley and Sons, 1965.

[13] C. W. Moon, P. R. Stephen and R. K. Brayton. Synthesis of hazard–free asynchronous circuits from graphical specifications. In *Proceedings of International Conference on Computer–Aided Design,* Pages 322–325, November 1991.

[14] T. Murata. Petri Nets: Properties, Analysis and Applications. *Proceedings of the IEEE*, 77(4): 541–580, April 1989.

[15] R. Puri. Signal Transition Graph Constraints for Hazard–free Synthesis of Asynchronous Circuits. *Unpublished manuscript*, June 1992. To be appeared in Proc. of ISCAS 93.

[16] S. H. Unger. *Asynchronous Sequential Switching Circuits.* John Wiley & Sons Inc., 1969.

[17] P. Vanbekbergen, F. Catthoor, G. Goossens and H. De Man. Optimized Synthesis of Asynchronous Control Circuits from Graph–theoretic Specifications. In *Proceedings of International Conference on Computer–Aided Design*, Pages 184–187, 1990.

[18] P. Vanbekbergen, F. Catthoor, G. Goossens and H. De Man. Time and area Performant Synthesis of Asynchronous Control Circuits. Technical report, IMEC Laboratory, B–3030 Leuven, Belgium, 1990.

[19] M.–L. Yu and P. A. Subrahmanyam. A New Approach for Checking the Unique State Coding Property of Signal Transition Graphs. In *Proceedings of the European Conference on Design Automation*, Pages 312–321, March 1992.