On the CSC property of Signal Transition Graph Specifications for Asynchronous Circuit Design

Mohit Sahni

Computer Science Department
Tokyo Institute of Technology
e-mail: msahni@cs.titech.ac.jp

Takashi Nanya

Research Center for Advanced Science
and Technology, University of Tokyo
e-mail: nanya@hal.rcast.u-tokyo.ac.jp

Abstract– This paper proposes a new approach for asynchronous logic synthesis from Signal Transition Graph (STG) specifications. The Complete State Coding (CSC) property of STGs is a necessary condition to get a circuit implementation from an STG. We present a novel method to check the CSC property of STGs. We also discuss some heuristics which automatically modify the STG so that the CSC property is satisfied. Our approach gives the designer some freedom to specify in what way a given STG is modified. Experimental results on a large set of benchmarks indicate a clear improvement over previous methods both in terms of time taken and in the reduction of the two level area literals.

I. Introduction

Recent progress in device technology is disclosing a fundamental restriction in synchronous system design. A natural solution to this problem is the introduction of asynchronous or event-driven computing[8]. Without a global clock, asynchronous circuit design can potentially solve many problems such as clock skew and power dissipation which are encountered by synchronous circuits. Speed-independent circuits are a class of asynchronous circuits that operate correctly in the presence of unbounded gate delays and zero wire delays. Signal Transition Graphs (STGs) are a subclass of interpreted Petri nets originally presented in [1] for the specification of speed-independent control circuits.

Synthesis of asynchronous circuits automatically from an STG specification has become an active research area[1, 10, 12]. This essentially consists of two steps: 1) Checking if a given STG satisfies conditions necessary for it to be implementable and 2) Logic Synthesis. The Complete State Coding (CSC)[5] properties, which guarantees that all the states in the specification are distinguishable, is one of the necessary conditions for an STG to be implementable.

There have been efforts to check the CSC property on a given STG and if the CSC property is violated, the STG is transformed to satisfy the CSC property. Initially the emphasis was on automatic repairing of STGs which did not satisfy CSC. Repairing was done either by introducing signals or adding more constraints in the form of arcs. A common characteristic of many of these schemes is that the State Graph (SG) derived from the STG is used as the main data structure. Algorithms in [4, 2, 13], all use the SG as an intermediate data structure. Since the size of an SG can be exponential on the number of signals in the circuit, all these methods show an exponential worst-case time complexity. Other methods use STG as the main data structure however, methods in [5, 9] can only handle STGs without choice operation (marked graphs). In [11], the method checks for Unique State Coding (USC) property which is a sufficient condition for the CSC property, so this method may modify an STG specification even when not required. In [7] an efficient algorithm for checking the CSC property was proposed however their method relies on only inserting new signals to modify the STG. None of the previous methods allows the designer to specify how an STG should be modified.

In this paper we propose a new algorithm which detects CSC violations. The algorithm works in the STG domain and can handle free choice nets. A net with a CSC violation is automatically modified. The modifications are made in the form of adding arcs and (or) inserting new signals, depending on the designer's priorities. Unlike previous methods our method gives the designer some freedom in the modification process. Experimental results show an improvement over the previous methods, in terms of computation speed and area of resulting circuits.

The paper is organized in the following way. Section II discusses some basic concepts of STGs. In section III we define our problem and Section IV discusses about our CSC verification algorithm, and the heuristics used to repair STGs with CSC violations. Experimental results and conclusions are given in sections V and VI.

II. Preliminaries

A Petri net(PN) is a four-tuple $\Sigma =< P, T, F, M_0 >$ where P, T and F form a directed bipartite graph. P represents a finite set of places and can specify choice or conflict. T represents a finite set of transitions. F gives the flow relation, $F \subset (P \times T) \cup (T \times P)$. A marking M

of a net is an assignment of non-negative integers called tokens to each place $p \in P$. M_0 defines the initial marking of the system.

A Petri net[1] is a very powerful way of describing the behaviour of concurrent systems[6]. When there exists a directed edge from transition t to a place p, t is called the *fanin* transition of p, and p is called the *fanout* transition of t. A *marked graph (MG)* is a net where every place has exactly one fanin and one fanout transition. A *state machine (SM)* is a net where every transition has exactly one fanin and one fanout place. A *free-choice (FC) net* is a Petri net where every place $p \in P$ with more than one fanout transition is the unique fanin place for all its fanout transitions. An FC net can be broken down into its MG components and SG components [1].

An STG is an *interpreted* net where each transition is interpreted as a physical transition of some signal. Signals can be of input or non-input (output and internal) signals. Input transitions come from the environment and in general it is assumed that the environment cannot be changed by the circuit. Transitions of signals are described by $s \times \{+,-\}$. $s+$ represents a rising transition of a signal s i.e the value of signal s is changing $0 \rightarrow 1$. $s-$ represents a falling transition of signal s. $s*$ denotes some transition of signal s (either $s+$ or $s-$), and $\overline{s*}$ denotes the complementary transition of $s*$. The State Graph (SG) is a finite automaton obtained by "executing" the STG. An STG is executed by examining all the possible markings reachable from M_0. Each node in an SG can be assigned a binary code as proposed in [1]. The binary code is simply the set of values that all the signals have in that particular state.

A. Properties of STGs

There are some important properties such as *correctness* and *complete state coding* which appear as syntactic conditions on the STG. Other properties of STGs are given in detail in [1].

Definition 1 *[1] A* **simple path** *in an STG is a sequence of transitions and places $\sigma = t_1 \ldots t_i t_j \ldots t_n$ s.t $t_i \neq t_j$, for any $i \neq j$. A* **simple cycle** *in an STG is a sequence of transitions and places $\phi = t_1 \ldots t_i t_j \ldots t_1$ s.t $t_i \neq t_j$, for any $i \neq j$.*

Definition 2 *A sequence SQ of transitions of an STG is said to be* **feasible**,*if there exists a path in the corresponding SG which exactly contains the transitions in SQ.*

Definition 3 *[3] An STG/FC is* **correct** *iff it satisfies the following four conditions:*

1. *In each feasible sequence of signal transitions, up and down transitions of the same signal alternate.*
2. *From any state, in any feasible sequence of signal transitions, the first change of certain signal is of the same sign (also called "initial stability").*
3. *No place in any reachable marking has more than one token.*
4. *Any free-choice place precedes only transitions of different input signals.*

Definition 4 *An STG is said to satisfy the* **Complete State Coding (CSC)** *property if,*

- *every state on its corresponding SG has a different binary code, OR*
- *when the same binary code is assigned for two different states S_1 and S_2, the enabled transitions for all the non-input signals are the same in both states S_1 and S_2.*

For a given net Σ, the *temporal Relation* is a symmetric binary relation applied to two items $n_1, n_2 \in T \cup P$. Let $n_1, n_2 \in T \cup P$ of a live FC net.

- n_1 and n_2 are ordered, $\{n_1, n_2\} \in li$, iff there is a simple cycle in Σ to which both n_1 and n_2 belong.
- n_1 and n_2 are concurrent, $\{n_1, n_2\} \in co$, iff $\{n_1, n_2\} \notin li$ and there exists an MG component of Σ to which both n_1 and n_2 belong.
- otherwise, n_1 and n_2 are in conflict, $\{n_1, n_2\} \in cf$,

Definition 5 *When a transition $o*$ immediately follows after another transition $t*$ and $\{o*, \overline{t*}\} \in co$, transition $o*$ is said to be* non-persistent. *An STG with a non-persistent transition does not satisfy Chu's persistency constraint[1].*

Correctness[2] and the CSC property are the necessary and sufficient conditions for the existence of implementation of circuits from the given STG. Correctness is required for preventing deadlock and the CSC property basically means that the behaviour of the physical circuit is the same in two states which have the same binary representation. The USC condition[9, 11] is only a easily verifiable sufficient condition for CSC. Chu's persistency is neither a necessary nor a sufficient condition for generating circuits from an STG[7].

III. Problem Definition

The property of correctness on an STG is not very difficult to check[3]. The CSC property is often violated in many STG specifications and is the most stringent requirement on STGs. High speed and efficient algorithms

[1] The terms *net* and *Petri net* are used interchangeably

[2] The condition of *liveness*, found in most STG literature is a slightly stronger condition than correctness

are desired for checking and solving the CSC property. Algorithms should avoid using the SG as an intermediate data structure due to the high cost factor in terms of time complexity. It is also desirable that the modifications made on the STG are in compliance with the requirements of the designer. Adding arcs to the STG may slow down the speed of the resulting circuit so arcs should be added only if they do not slow down the circuit beyond the designer's specified limit.

IV. Methodology

Starting from a given STG, state codes with dont-cares are assigned to each place in the corresponding Petri net. In any reachable marking of the net, the set of state codes corresponding to the places in which a token is present, gives complete information about the state of the system. Thus, a CSC conflict in the SG will correspond to two different markings of the net where the state codes have the same value. The proposed algorithm finds out pairs of markings which cause a CSC conflict. Then, we use two heuristic approaches to solve the CSC conflict. The first heuristic adds arcs in the STG to remove some CSC conflict causing states. The second heuristic adds new signals to make the STG CSC conflict free.

A. The CSC Property: Verification

Fig. 1(a) shows an STG which has a CSC conflict. The conflict occurs because there exists a *complementary path* $\{a+, a-\}$ in the STG. A *complementary path* in an MG component is a feasible sequence SQ of transitions of the STG, such that $SQ \neq T$ and there are an equal number of rising and falling transitions for each signal. In an MG, for every CSC violation there exist two complementary paths P_1 and P_2 such that $P_1 \subset MG$, $P_2 \subset MG$, $P_1 \cap P_2 = \emptyset$ and $MG = P_1 \cup P_2$.

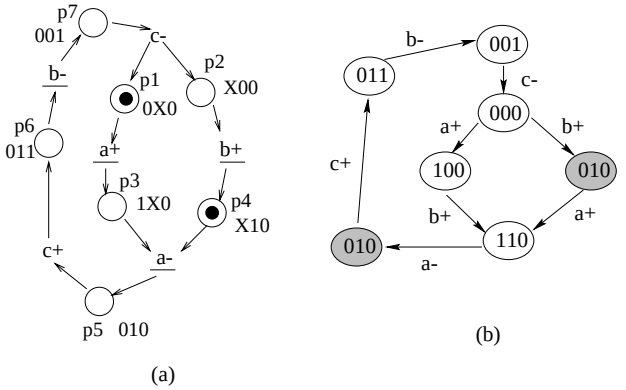

Fig. 1. CSC Violation (a) STG with State Codes (b) State Graph

Definition 6 *A CSC conflict is said to be a* **unique CSC conflict** *if, of the two complementary paths corresponding to the CSC conflict, at least one cannot be broken down into smaller complementary paths.*

A.1. Assigning State Codes

For assigning the state codes we use the method described in [12]. We use a relation called the *Interleaved relation.*

Definition 7 *Let a*/i and a*/j be two different transitions of the same signal a. A place p is said to be* **Interleaved** *with transitions a*/i and a*/j if there exist simple paths σ_1 and σ_2 such that*

- *a*/i $\sigma_1 p \sigma_2$ a*/j forms a simple path,*
- *σ_1 and σ_2 don't contain any transition of signal a, and*
- *there is no transition of signal a which is concurrent to any transition $t \in \sigma_1 \cup \sigma_2 \cup \{p\}$.*

For example in Fig. 1 it can be seen that place p_1 is interleaved between $a-$ and $a+$. Similarly place p_5 is interleaved between $b+$ and $b-$.

Every place p in a given petri net Σ can be assigned a code C_p. The code has a literal corresponding to each signal in Σ. Let C_p^a denote the literal corresponding to signal a. Then,

$$C_p^a = \begin{cases} 1 & \text{if } p \text{ is interleaved with } (a+/*, a-/*) \\ 0 & \text{if } p \text{ is interleaved with } (a-/*, a+/*) \\ X & \text{if } p \text{ is concurrent with some transition of } a \\ - & \text{otherwise} \end{cases}$$

The *sum* of two literals l_1 and l_2 is defined as l_1 if $l_1 = l_2$ or $l2 = X$, l_2 if $l_1 = X$ else *sum* of l_1 and l_2 is undefined. The *sum* of two states codes C_{p1} and C_{p2} of places $p1$ and $p2$ of the Petri net is defined as the sum of each of its corresponding literals. If the sum of any of their corresponding literals is undefined, the sum of C_{p1} and C_{p2} is undefined.

Theorem 1 *If $p1, p2 \in P$ and $\{p1, p2\} \in co$ and Σ is a correct FC net, sum(C_{p1}, C_{p2}) is always defined.*

Proof: Follows from definition of correctness in [3].

From Theorem 1 it follows that the sum *(or code)* of a (all the places in) a marking is always defined for a correct STG. In Fig. 1, the code of marking $\{p1, p4\}$ is the sum of the places having tokens in the marking. Taking the sum of state codes for places $p1$ and $p4$, we get state $0X0 + X10 = 010$.

Definition 8 *A marking is said to be* **partial** *if it does not completely specify the state of the system otherwise it is* **complete**.

It is sufficient to check the STG/FC net for unique CSC conflicts. If the unique CSC conflicts are removed the STG/FC net can satisfy the CSC property. For each unique CSC conflict we can find two (partial or complete) markings M_1 and M_2 which have exactly the same

code. For Fig. 1, $M_1 = \{p1, p4\}$ and $M_2 = \{p5\}$ and $Code(M_1) = Code(M_2) = 010$. However, if M_1 and M_2 are partial markings with same codes and if all the places which are concurrent with all the places in M_1 are the same as all the places concurrent with all the places in M_2, a CSC conflict occurs. All it means is that a certain set of places can be added to both M_1 and M_2 to complete their markings but the actual CSC conflict is caused by the transitions fired between M_1 and M_2 which form a complementary set. Fig. 2 shows a part of an STG. Places p_4, p_5, p_6 and p_7 all are concurrent with places p_1, p_2 and p_3. $\{p_1\}$ specifics a partial marking which has same code as partial marking $\{p_3\}$ and this is sufficient to detect the CSC conflict caused by the transitions $\{a+, a-\}$. We call the pair of markings p_1, p_2 and p_3, as **minimal** as it is enough to specify the CSC violation.

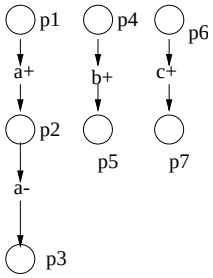

Fig. 2. Part of an STG used to show sufficiency of Partial Marking

Lemma 1 *Let* $M_1 = \{p_{1^1}, p_{1^2}, \ldots, p_{1^n}\}$ *and* $M_2 = \{p_{2^1}, p_{2^2}, \ldots, p_{2^m}\}$ *be two minimal (partial or complete) markings which cause a unique CSC conflict. Let* $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ *be a graph such that* $\mathcal{V} = M_1 \cup M_2$ *and* $v_i \in \mathcal{V} \wedge v_j \in \mathcal{V} \wedge \{v_i, v_j\} \in li$, $\{v_i, v_j\} \in \mathcal{E}$. *Graph* $\mathcal{G}$ *is connected.*

Proof: The above lemma is proved by contradiction. Let us assume that the graph is not connected and let G_1 be a subgraph. If G_1 has only one node n then, n is not connected with any other node and n is concurrent to all nodes in $\mathcal{V} - \{n\}$. This means that n is a redundant node which appears in both the given markings and the markings M_1 and M_2 are not minimal, which is a contradiction. Next if G_1 has more than one nodes then, G_1 contains nodes both from M_1 and M_2 (as nodes only in M_1 or M_2 cannot be connected). Let $N_1(N_2)$ represent the nodes from $M_1(M_2)$ which are in G_1. All nodes in G_1 will be concurrent to all nodes in $\mathcal{V} - N_1 \cup N_2$. This means that all the transitions between N_1 and N_2 (or N_2 and N_1) should be concurrent to all nodes in $\mathcal{V} - N_1 \cup N_2$. Since M_1 and M_2 cause a CSC conflict and two transitions of the same signal cannot be concurrent to each other, N_1 and N_2 also cause a complementary path, which means that the CSC conflict specified by M_1 and M_2 is not unique. Hence a contradiction. □

Figure 3 shows the connected graph component got from the conflicting markings $\{p_1, p_4\}$ and $\{p_5\}$ got from the STG in Fig.1a.

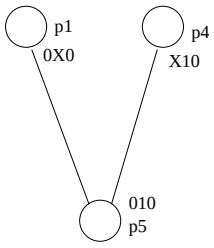

Fig. 3. The connected graph component $\mathcal{G}$ for the STG in Fig.1a

A.2. The Algorithm

Lemma 1 is the basis for our algorithm. So the problem of detecting CSC conflicts reduces to finding such a connected graph which specifies the conflict causing markings. For a unique CSC conflict, each pair of nodes that is connected in the graph have **matching** codes. We say that the codes of two places are said to **match** if each literal in the two codes matches. A dont-care X matches 0,1 or X. Two codes are said to be **equal** if each literal in a code is the same as the corresponding literal in the other code. So after forming a set of all the pair of places with matching codes (which can be done in $O(N^2)$ time, N being the number of transitions in the given net), we use a simple *Expand* algorithm to find the CSC conflict. Our *Expand*[3] algorithm starts with a matching pair and keeps on adding matching pairs recursively until all the possible markings which can cause a possible CSC conflict are formed. For each pair of matching nodes our algorithm takes $O(N)$ time. So for all the pairs the time taken is $O(N^2)$. So our algorithm detects a CSC violation in $O(N^2)$ time.

The CSC Algorithm has two parts based on the following. Let S_1 and S_2 be two different states in a SG. Let T_1 be the set of transitions which lead to state S_1 and T_2 be the corresponding set for s_2. The following two cases may arise. Either,

1. all transitions in T_1 are in conflict with all transitions in T_2, or
2. otherwise.

The first case means that the two states can only be reached by conflicting transitions; i.e. they can never be reached by one MG component we call these as the *Conflicting States*. The second case means that both states s_1 and s_2 can be reached by one MG component. We call such states as the *Non-conflicting States*. The first part of

[3] We donot claim that our *Expand* algorithm is the most efficient way to find the connected graph (Lemma 1), but it is fast and simple.

the algorithm deals with checking CSC on non-conflicting states and the second part covers the conflicting states. For conflicting states the graph $\mathcal{G}$ (Lemma 1) which causes a CSC violation is connected but an edge occurs between two nodes only if they are in conflict.

Algorithm:

For non-conflicting state codes

begin

1 Make a match set $M = [< p_1, p_2 >: \{p_1, p_2\} \in li \wedge p_1$ *matches* $p_2]$

2 While $M \neq \phi$ *do*

2.1 Take one element $< p_1, p_2 >$ *out of* M*, set* $P_1 = \{p_1\}$ *and* $P_2 = \{p_2\}$

2.2 Expand(P_1, P_2) *recursively. For* $< p_i, p_j > \in M \wedge p_i \in P_1(P_2) \wedge \forall p_k \in P_2(P_1)$, $\{p_j, p_k\} \in co$, *then* $< p_i, p_j >$ *can be used to expand* P_1 *or* P_2*. Expand recursively adds elements to* P_1 *or* P_2 *until none of them could be expanded any furthur.*

2.3 If codes represented by P_1 *and* P_2 *are not equal, goto Step 2.*

2.4 If codes represented by P_1 *and* P_2 *are equal and have don't cares then*

2.4.1 Let $E_1(E_2)$ *denote the set of places which are concurrent to each place in* $P_1(P_2)$

2.4.2 If $E_1 \neq E_2$ *then goto Step 2.*

2.5 If the non-input signals enabled by set P_1 *are the same as enabled by set* P_2 *then goto Step 2.*

2.6 Report: CSC not satisfied by sets P_1 *and* P_2*, goto Step 2.*

end

For conflicting state codes

begin

1 Make a match set $M = [< p_1, p_2 >: \{p_1, p_2\} \in cf \wedge p_1$ *matches* $p_2]$.

2 same as Step 2 above.

end.

B. The CSC Property: Repairing

In this section we will discuss two heuristics which attempt to solve a CSC violation in an STG/FC net.

B.1. Heuristic 1

Fig. 4 shows a part of an STG. t_1 and t_2 are concurrent in the original STG but if a constraint $t_1 \rightarrow t_2$ is added, the circuit speed might get slower. Some of the transitions which follow t_2 and are concurrent to t_1 will also have to wait for t_1 to fire. t_3 represents such a transition which is the farthest (i.e t_3 does not lead to any transition which is both concurrent to t_1 and dependent on t_2). t is the closest transition which is ordered with both t_1 and t_2 (t can be found by simple back tracking from t_1 and t_2). d_1, d_2 and d_3 represent the distances (number of transitions) in the respective paths.

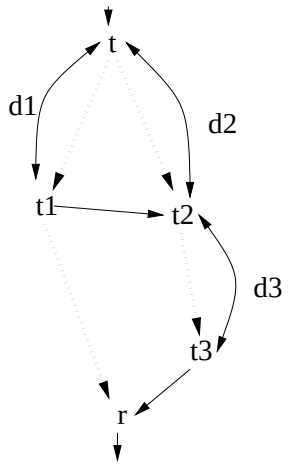

Fig. 4. Cost of adding $t_1 \rightarrow t_2 = d_1 + d_3 + 2 - d_2$

Now lets take the worst case that can occur to degrade the circuit performance. After the firing of t if all the transitions between t and t_2 have fired and none of the transitions between t and t_1 have fired. This delays the firing of the furthest transition t_3. It would have been possible that without the arc $t_1 \rightarrow t_2$, transition t_3 would have fired before the firing of t_1. So now the extra delay that occurs is equivalent to firing of $d_1 + d_3 + 2$ transitions (the factor 2 is to incorporate the transitions t_1 and t_2). However this estimate is too pessimistic. In a real circuit if d_2 has a high value, it is less probable that such a case occurs (assuming that none of these include time consuming operations). A better esimate would be that since transitions between t and t_1 occur concurrently with transitions between t and t_2, $d2$ transitions should be subracted from d_1 transitions. This gives the cost as a function $Cost(t_1 \rightarrow t_2) = d_1 + d_3 + 2 - d_2$. The *Cost* function simply estimates the number of transitions that a particluar transition can be delayed due to the adding of arc $t_1 \rightarrow t_2$.

To compare two different costs we assume that the more the number of transitions in a path the more the worst-case delay[4]. The *Cost* function is used to choose between different candidates and it is also used for interacting with the designer. The designer can specify the acceptable loss of concurrency and then no arc will be added if the cost is more than what is specified by the designer. The user may also set the acceptable loss of concurrency to 0 or ∞ if required.

Many of the CSC conflicts can be solved by just adding one arc. The arc can be added as shown in Fig. 5. The

[4] The delay of a transition highly depends on the number of fanins and fanouts. So, in a true situation the worst-case delay of one transtion may be more than the sum of the worst-case delay of two transitions.

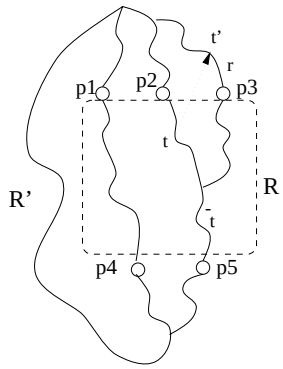

Fig. 5. Adding one arc to solve a CSC violation

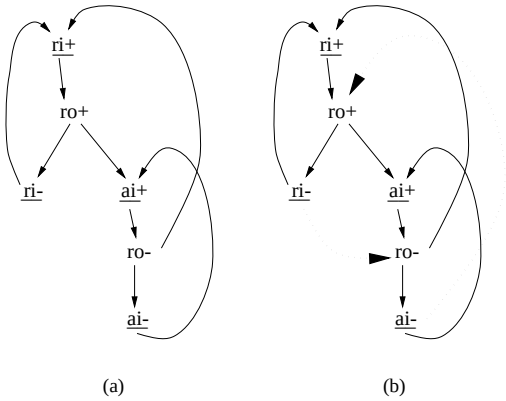

Fig. 6. (a) An STG from the SIS benchmarks (b) CSC violation removed

two marking $\{p_1, p_2, p_3\}$ and $\{p_4, p_5\}$ have the same state code and cause a CSC violation. If two places which belong to one of the two markings ($\{p_1, p_2, p_3\}$ or $\{p_4, p_5\}$) become ordered after adding some arc, that marking is never possible again. Consider Fig. 5, if an arc is added from transition t to transition t', the two places p_2 and p_3 which were concurrent earlier become ordered. So the marking $\{p_1, p_2, p_3\}$ can never occur again and this solves the CSC conflict.

But adding arcs may increase the circuit size or make logic synthesis difficult. In [10], it was shown that in the synthesis of circuits from STGs which donot satisfy Chu's persistency constraint require an OR-gate in their synthesis. Though the circuits are speed-independent, they are more prone to hazards when realized actually where wire do have some delays. Thus we try that we donot introduce Chu's non-persistency while adding arcs.

So for adding an arc from transition t to t' which are concurrent transitions the following constraints must be satisfied.

1. *t and t' should belong to separate complementary regions, say $\mathcal{R}$ and $\mathcal{R}'$*
2. *t' shouldn't be a transition of an input signal.*
3. *After putting the arc there should be an order $t \rightarrow t' \rightarrow \bar{t}$. This guarantees that there is least overhead in terms of circuit size and also Chu's persistency is not violated.*
4. *$t \rightarrow t'$ should make two places of a CSC conflict causing marking to be in order (putting the arc should atleast solve one CSC conflict)*
5. *Cost($t \rightarrow t'$) should be less that overhead allowed by designer.*
6. *Putting this arc should not put time consuming concurrent operations in order.*

Arcs are added iteratively using a greedy approach. Adding arcs for conflicting states is also done in a simliar manner. As an example consider the STG in Fig. 6(a). The CSC violation occurs there is a complementery path $\{a_i-, a_i+\}$. If the violation is to be solved by insering signals, no signal can be inserted in this complementary path as it has only input transitions. Signals can be inserted to make the non-input signals enabled in the conflicting state to be the same but this may require many new signals. However, the same problem can also be solved by inserting arcs. Fig. 6(b) shows the solution by adding these arcs. The cost of adding these arcs is also minimal.

B.2. Heuristic 2

Heuristic 1 given in the presvious section is very strict and in most cases it is not able to remove all CSC conflicts. In such a case we remove CSC conflicts by inserting signals. The adding of signals is done in the following manner.

1. *For each conflict find the complementary paths from the pair of markings given by our Algorithm. Also make a kist of all the possible places where a signal can be inserted.*
2. *Then we choose two points which are ordered and which solve a maximum number of CSC conflicts and these add transitions $s+$ and $s-$ at these points (s is a new signal)*
3. *Step 2 is repeated till all the conflicts are removed.*

V. Experimental Results

Table V shows the results of the proposed algorithm. The algorithm was implemented in the C language and the program was run on a Sun Sparc Workstation. We tested our method on a large number of benchmarks and we compared our results with those of SIS[4] and Petrify[2]. Our algorithm outperforms both SIS and Petrify both in terms of time taken and in reducing the two

level area literals. For heurestic 1 we did not use any limit on permissible reduction in concurrency. Just for comparison, Petrify took over 80 sec to solve the CSC for benchmark mr1 while, our method takes less than a second.

TABLE I
EXPERIMENT RESULTS

STG Name	Specifications initial			SIS final		Petrify final		Our Method final		
	sig.	trans.	states	sig	lit	sig	lit	sig	lit	time (s)
par_4	10	28	628	Error		14	41	14	41	0.64
mr0	11	22	302	13	86	14	53	14	42	0.94
mr1	9	18	190	11	53	12	43	11	36	0.57
mmu0	8	16	174	Error		11	46	9	23	0.46
mmu1	8	16	82	10	37	10	32	8	21	0.57
sbuf-ram-write	10	20	58	12	35	12	24	11	34	1.11
vbe4a	6	12	58	8	41	8	24	9	29	0.25
nak-pa	9	18	56	10	41	10	21	10	19	0.51
ram-read-sbuf	10	20	36	11	23	11	20	10	21	0.59
sbuf-send-pkt2	6	23	21	7	14	7	18	7	15	0.55
duplicator	4	12	20	5	24	6	22	5	12	0.26
sbuf-read-ctl	6	12	14	7	15	7	15	7	16	0.22
seq8	18	36	36	Error		24	49	22	45	2.54
seq_mix	8	20	20	10	34	11	23	10	24	0.53
spec_seq4	10	20	20	12	37	13	23	12	23	0.50
atod	6	12	20	7	19	7	19	7	12	0.23
alloc-outbound	7	20	17	9	23	9	18	9	16	0.47
TOTAL					482		491		429	10.94

VI. Conclusions and Future Work

Efficient methods for the synthesis of asynchronous logic from STG/FC net specifications are presented. Our algorithm checks for the CSC property on FC nets and is efficient on computing time. Unlike most of the previous methods, we work directly on the STG and obviate the need for an SG representation. We have shown that how the pair of markings which lead to a CSC violation form a connected graph and our method basically works on finding such a connected graph.

FC nets which violate the CSC property are repaired using two heuristic techniques. The first technnique adds arcs to serialize some concurrent operations. Arcs are added only if they also reduce the circuit size while not reducing the speed of the resulting circuit considerably. We have tried to associate a cost factor which gives an idea of how much concurrency is lost if an arc is added. Previous methods have never allowed the designer to have any say in the modification process. The second heuristic simply adds new signals to remove the CSC violations. Compared to previous methods our approach leads to a clear improvement in terms of both compution speed and circuit areas.

Future work includes algorithm improvement and better repairing techniques. Compution speed can be improved furthur by using a better algorithm (rather than our *Expand*) for finding the graph which causes the CSC conflict. Better and more realisitic ways of measuring the cost of adding an arc in the STG are also required. As far as we know till now there hasn't been much work on analyzing the cost in terms of loss of concurrency.

References

[1] Tam-Anh Chu. *Synthesis of Self-timed VLSI Circuits from Graph-theoretic Specification*. PhD thesis, MIT, June 1987.

[2] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev. Methodolgy and Tools for State Encoding in Asynchronous Circuits. In *Proc. of 33rd DAC*, pages 63–66, 1996.

[3] A. Kondratyev and A. Taublin. On Verification of the Speed-Independent Circuits by STG unfoldings. Technical Report 94-2-001, The University of Aizu, 1994.

[4] L. Lavagno, C.W. Moon, R.K. Brayton, and A. Sangiovanni-Vincentelli. Solving the State Assignmnent Problem for Signal Transition Graphs. In *Proc. of 29th DAC*, 1992.

[5] K.J. Lin and C.S. Lin. Automatic Synthesis of Asynchronous Circuits. In *Proc. of 28th DAC*, pages 296–301, 1991.

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

[7] R. Nagalla and G. Hellestrand. Signal Transition Graph Constraints for the Synthesis of Hazard-free Asynchronous Circuits with Unbounded-Gate Delays. In *Formal Methods in System Design, 5*, pages 245–273, 1994.

[8] T. Nanya. Challenges to dependable Asynchronous Processor Design. In *Int'l Symp. on Logic Synthesis and Microprocessor Architecture*, pages 132–139, July 1992.

[9] F. Cathoor P. Vanbekbergen, G. Goosens and H. J. De Man. Optimized Sythesis of Asynchronous Control Circuits from Graph-Theoritic Specifications. *IEEE Transactions on CAD*, 11(11), November 1992.

[10] S. Park and T. Nanya. Synthesis of Asynchronous Circuits from Signal Transition Graph Specifications. *IEICE Trans. Information Systems*, E80-DI(3):351–361, March 1997.

[11] E. Pastor and J. Cortadella. An Efficient Unique State Coding Algorithm for Signal Transition Graphs. In *Proc. of ICCD*, pages 174–177, 1993.

[12] E. Pastor, J. Cortadella, A. Kondratyev, and O. Roig. Structural Methods for the Synthesis of Speed-Independent Circuits. In *Proc. of European Design and Test Conference*, pages 340–347, 1996.

[13] P. Vanbekbergen, B. Lin, G. Goosens, and H. J. De Man. A Generalized State Assignment Theory for Transformation of Signal Transition Graphs. In *Proc. of ICCAD*, pages 184–187, 1992.