A Divide-and-Conquer Approach for Asynchronous Interface Synthesis †

Ruchir Puri and Jun Gu
Dept. of Electrical & Computer Engineering
University of Calgary, Calgary, Canada T2N 1N4

Abstract

Asynchronous circuits are crucial in designing low power, high performance digital systems. They are widely used in many real time applications such as digital communication and computer systems. The design of complex asynchronous interface circuits is a difficult and error-prone task. In this paper, we present an area and time efficient synthesis algorithm for general signal transition graph (STG) specifications. It utilizes a divide-and-conquer approach to significantly reduce the number of design constraints. Thus, large size specifications can be synthesized in a very short execution time. We further developed a BDD based constraint satisfaction algorithm that exploits the don't cares for area efficient synthesis. Experimental results with a large number of practical signal transition graph benchmarks are presented. These results show that compared to the existing techniques, the divide-and-conquer technique is capable of achieving an average of 20% reduction in implementation area for all the benchmarks and it offers a practical solution for the complex asynchronous interface design problems.

1 Introduction

Asynchronous interface circuits are indispensable in many real-time digital systems. An asynchronous interface combines various microprocessors, RAMs, and ASICs which operate either at different clock rates or in an asynchronous way. Due to the potential applications in lower power and speed-independent systems, recently, there has been a renewed interest in the automated synthesis of asynchronous interface circuits [3, 15, 25]. The design of asynchronous control and interface circuits is a complex and error prone job. An efficient method to synthesize interface circuits will greatly simplify their design and reduce the design errors. Previous researchers have developed an event based graphical specification [3] and a direct synthesis method [3, 25, 27]. They are unable to synthesize complex designs involving large number of constraints. The divide-and-conquer synthesis technique proposed in this paper offers a practical solution to solve the complex design problems.

The event based graphical specifications, also called signal transition graphs (STG) [3], are based on petri nets [17]. They are very powerful in describing the major aspects of asynchronous interface circuit behavior, such as concurrency, conflict (i.e., choice), and causality (i.e., sequencing). The problem of synthesizing asynchronous interface circuits from STG specifications has been studied by many researchers [3, 15, 25, 26, 27]. To implement the asynchronous behavior into a logic circuit, the STG specifications must satisfy liveness, boundedness, and complete state coding (CSC) constraints. Informally, liveness implies the absence of deadlocks in a circuit. Boundedness ensures that the system is finite. The CSC constraint implies that the signals specified by the STG completely define the circuit states.

Complete state coding is the most stringent requirement on STG specifications. In general, the STG specifications that describe practical asynchronous interface circuits do not satisfy the CSC constraints and the original STG must be transformed to satisfy it. Most methods are limited by the type of asynchronous interface behaviors they can synthesize. Lin et al. [15], Vanbekbergen et al. [25], and Yu et al. [27] proposed synthesis techniques that are restricted to the STG specifications describing only concurrent asynchronous behavior (marked graphs). These techniques were limited by an additional restriction that every signal has exactly one rising and falling transition. This severely limits the type of practical asynchronous behaviors that can be synthesized. Lavagno and Moon et al. [14] proposed a new synthesis framework for the STG specifications with a limited interplay of both concurrency and choice (safe free-choice nets). They solved the STG synthesis problem at the state graph level by transforming the STG into an FSM state table. The synthesis problem is then solved using state minimization [21] and critical race-free state assignment [23]. This technique inserts state signals into the original STG to satisfy CSC constraints. It handles live-safe free choice nets and guarantees sufficient conditions for the CSC-satisfaction. The CSC solutions obtained in this framework correspond to a special class of STG transformations.

Recently, Vanbekbergen et al. [26] proposed a framework to solve the complete state coding problem for general STG specifications. It is not limited to marked graph or safe free-choice petri nets. They formulated the CSC problem as a boolean satisfiability (SAT) problem[1]. They gave the necessary and sufficient conditions for the insertion of state signals. This ensures the CSC property while conserving the original STG behavior. It is well-known that many combinatorial optimization problems can be directly transformed into boolean satisfiability problem. Unfortunately, the instances of satisfiability formulas derived from the practical STGs are too large to be solved efficiently. For example, a signal transition graph [18] with 302 states generates a boolean formula with 1,208 variables. In our experience [6, 7, 8, 9, 10, 12, 11, 20], it usually takes prohibitively long time to find a satisfiable assignment for very large boolean formulas. In addition, Vanbekbergen et al.'s synthesis framework minimizes the number of state signals required to satisfy CSC property. The SAT algorithm employed in this framework can yield only one solution. The solution is often very inefficient in terms of circuit complexity [24]. A SAT framework that enables the efficient enumeration of several solutions can yield reduced circuit complexity.

In this paper, we propose a divide-and-conquer synthesis technique for asynchronous interface circuits. We solve the CSC problem by satisfying the synthesis constraints at the decomposed signal transition graphs level, rather than at the original STG specification level. Decomposing the state

†This research is supported in part by the 1993 ACM/IEEE Design Automation Award, by the Alberta Microelectronics Graduate Scholarship, by the NSERC research grant OGP0046423, and was supported in part by the NSERC strategic grant MEF0045793.

[1]The boolean satisfiability problem is the problem of finding a truth assignment to variables in a given product-of-sums expression, so that the boolean expression evaluates to be *true*. SAT was the first problem to be proven NP-complete [4].

0-8186-5785-5/94 $03.00 © 1994 IEEE

graph avoids the problem of solving a very large instance of the boolean satisfiability formula. Our synthesis technique is applicable to general STGs and is not limited to marked graph or safe free-choice petri nets. This approach outperforms both Lavagno et. al.'s [14] and Vanbekbergen et al.'s [26] algorithms in terms of the execution time. In our synthesis framework, the state graph constraints are satisfied by developing a Binary Decision Diagram (BDD) [2] constraint satisfaction algorithm. A simple traversal of BDD paths enables efficient enumeration of several solutions. Thus, circuits with reduced circuit complexity can be obtained. Our algorithm yields an average of 20% reduction in two-level implementation area with respect to both the Lavagno et al.'s and Vanbekbergen et al.'s algorithms for all the circuits in the asynchronous circuit benchmarks [14].

The rest of this paper is organized as follows. In Section 2, we discuss some basic definitions and notations that simplify our discussion. In Section 3, we describe our divide-and-conquer synthesis algorithm in detail [22]. Experimental results with practical asynchronous STG benchmarks, including the benchmark set of [14], are illustrated in Section 4. Section 5 concludes this paper.

2 Preliminaries

A *petri net* [17] is a bipartite directed graph $< P, T, F, M_0 >$, consisting of a finite set of *transitions* T (represented as bars), a finite set of *places* P (represented as circles), and a *flow relation* $F \subseteq P \times T \cup T \times P$ (represented as directed arcs) specifying a binary relation between transitions and places. The dynamic behavior of a petri net is captured by its *markings* and the *firing* of net transitions, which transforms one marking into another. A marking M is a collection of places corresponding to the local conditions which hold at a particular moment. It is graphically represented as solid circles called *tokens*, residing in these places. The *initial marking* is denoted as M_0. A transition t is said to be *enabled* in a marking M, when all its fanin places are marked with at least one token. An enabled transition must eventually fire and its firing removes one token from each fanin place and deposits one token in each fanout place. The transformation of a marking M into another marking M', by firing a transition t is denoted by $M \xrightarrow{t} M'$.

Signal transition graphs use petri nets as their underlying formalism to specify the behavior of digital control circuits. In an STG, petri net transitions are interpreted as rising and falling transitions in the asynchronous interface circuit. Transitions s_i+, s_i-, and $s_i\bullet$ denote a rising, a falling, and a rising or falling transition on signal wire s_i. The set of input signals and the set of non-input, i.e., output and internal signals, is denoted by S_I and S_{NI}, respectively. The set of all the signals in the STG, i.e., $S_I \cup S_{NI}$, is denoted by S. In an STG, every place with a single fanin and fanout transition is represented by an arc between these transitions.

A signal transition graph contains the behavioral information of an asynchronous interface circuit. To derive a logic circuit, the STG must be transformed into a state graph [3]. The state graph is a finite automaton representing all the states. It captures all the possible transition sequences in the STG. A state graph can be derived by exhaustively generating all possible markings, i.e., states of the STG [17]. A state graph can be mapped into a circuit by satisfying CSC constraints, i.e., assigning a unique binary code to each state in the state graph. The state graph can be mapped into a speed-independent asynchronous logic circuit by deriving the state code from the values of STG signals. Such a state encoding is represented by the consistent state assignment constraint.

Consistent state assignment : For STG signals $\{s_1, s_2, \ldots, s_n\}$, a state M in the state graph is assigned a binary code $< M(s_1), M(s_2), \ldots, M(s_n) >$. If a transition t is enabled in state M, i.e., $M \xrightarrow{t} M'$, then $t = s_i+$ implies $M(i) = 0$ and $M'(i) = 1$; $t = s_i-$ implies $M(i) = 1$ and $M'(i) = 0$.

In a state graph with consistent state assignment, the CSC constraint is defined as follows.

Complete state coding (CSC) : A state graph satisfies the CSC constraint if and only if (1) no two states have the same binary code assignment, and (2) two states having the same binary code enable the same non-input signals.

Moon et al. [16] showed that the CSC constraint is the necessary and sufficient constraint to derive the logic circuit functions from the state graph. A CSC violation must be corrected by inserting new signals in the state graph, so as to distinguish between the states violating CSC constraint [14, 26]. These new signals are called *state signals*. They must satisfy an additional constraint called semi-modularity constraint to preserve the given circuit behavior. The semi-modularity constraint is defined as follows.

Semi-modularity : A transition t is *semi-modular* if and only if transitions t and t' are enabled in a marking M and transition t will still be enabled in marking M', obtained after firing transition t'.

The insertion of state signals in the state graph must preserve the semi-modularity of the state graph transitions. The most general framework to solve the CSC problem for general STG specifications by inserting *state signals* into the state graph was proposed by Vanbekbergen et al. [26]. They formulated the CSC problem as a boolean satisfiability (SAT) problem. In the following section, we briefly describe the SAT formulation of the CSC constraint satisfaction problem.

2.1 A SAT model for CSC-satisfaction

The satisfiability (SAT) model for CSC satisfaction, i.e., SAT-CSC, has four components :

- A set of N *states* in the state graph: $M_1, M_2, \ldots, M_N$.
- A set of m *state variables*: a state M_i with m state signals $n_1, n_2, \ldots, n_m$ is denoted by $M_i\{n_{i,1}\}\{n_{i,2}\}\ldots\{n_{i,m}\}$.
- A four *value* tuple with possible assignments to the state variables: {0, 1, *Up*, *Down*}.
- A set of *constraints* including CSC constraints, consistent state assignment constraints, and semi-modularity constraints (as defined above).

The SAT-CSC model has N states and $N.\lceil log_2(Max_{csc})\rceil$ state variables.[2] The number of state variables, $m = \lceil log_2(Max_{csc})\rceil$, is the *lower bound* on the number of state signals required to satisfy the CSC constraints, where Max_{csc} denotes the maximum number of states in the state graph that has the same state encoding.

The boolean constraint formula represents the CSC, consistent state assignment, and semi-modularity constraints. If the formula is unsatisfiable, we add a new state signal. This generates a new constraint formula. The *goal* is to find a truth

[2]Since, a state variable $n_{i,k}$ is a multi-valued variable, its boolean encoding requires $\lceil log_2(4)\rceil = 2$ binary variables, denoted as n_{i,k_a} and n_{i,k_b}. The binary variable assignments $\{n_{i,k_a} = 0, n_{i,k_b} = 0\}$, $\{n_{i,k_a} = 0, n_{i,k_b} = 1\}$, $\{n_{i,k_a} = 1, n_{i,k_b} = 0\}$, and $\{n_{i,k_a} = 1, n_{i,k_b} = 1\}$ represent the state variable $n_{i,k}$ values 0, 1, *Up*, and *Down*, respectively.

assignment to the state variables so that the constraints are satisfied. We use a simple example, *PaBlock* (Figure 1(a)), to illustrate the method.

Example : The state graph of the example STG is shown in Figure 1(b). It is derived by exhaustively generating all the 18 states of the STG *PaBlock* [17]. The state graph contains 10 state pairs with the same state encoding, i.e., the state sets $\{M_{17}, M_{11}, M_5\}$, $\{M_{15}, M_9, M_2\}$, $\{M_{13}, M_7, M_1\}$, and $\{M_{12}, M_4\}$ have the same binary codes 0010, 1000, 0000, and 1001, respectively. Since the non-input signals enabled in state M_{17} and state M_{11} are the same, i.e., $po-$, these states do not violate the CSC constraint (as defined above). Similarly, states M_{13} and M_1 do not violate the CSC constraint. Thus, there are 8 CSC constraints in the state graph, i.e., $\{M_{17}, M_5\}$, $\{M_{11}, M_5\}$, $\{M_{15}, M_9\}$, $\{M_{15}, M_2\}$, $\{M_9, M_2\}$, $\{M_{13}, M_7\}$, $\{M_7, M_1\}$, and $\{M_{12}, M_4\}$.

Since the maximum number of states with the same binary encoding $< 1000 >$ is *three*, e.g., $\{M_{15}, M_9, M_2\}$, the lower bound on the number of state signals is $\lceil log_2(3) \rceil = 2$. Thus, while keeping the consistent state assignment and state graph semi-modularity, the SAT-CSC model requires *two* state signals, n_1 and n_2, to satisfy the CSC constraints. Two state variables $n_{i,1}$ and $n_{i,2}$, corresponding to the state signals n_1 and n_2, are assigned to every state M_i in the state graph. This is represented by $M_i\{n_{i,1}\}\{n_{i,2}\}$ in the final state graph in Figure 1(e).

The constraints can be formulated as boolean relations on the binary variables $n_{1,1_a}, n_{1,1_b}, n_{1,2_a}, n_{1,2_b}, \ldots, n_{18,2_b}$ corresponding to the state variables $n_{1,1}, n_{1,2}, \ldots, n_{18,1}, n_{18,2}$, respectively. This model requires 72 binary variables in the constraint formula, i.e., for state variables $n_{i,1}$ and $n_{i,2}$, four binary variables $n_{i,1_a}, n_{i,1_b}, n_{i,2_a}, n_{i,2_b}$ are required for each state M_i. The final assignment of the state variables resolves all the CSC conflicts in the state graph in Figure 1(e). For example, the CSC conflict between states M_2 and M_9 is resolved by the state variables $n_{2,1}$ and $n_{9,1}$. State variable $n_{2,1}$ assigns value 0 to state M_2 and state variable $n_{9,1}$ assigns value 1 to state M_9. This is denoted by $M_2\{0\}\{D\}$ and $M_9\{1\}\{0\}$ in Figure 1(e). The state graph with final state variable assignments is shown in Figure 1(e).

The following section describes our divide-and-conquer approach for asynchronous interface circuit synthesis. Throughout, we use the STG example *PaBlock* to illustrate this approach.

3 A Divide-and-Conquer Approach

Most of the techniques proposed for asynchronous circuit synthesis are restricted in practical applications [15, 25, 27]. Some of them try to satisfy all the constraints in the state graph directly, which, in most cases, is clearly intractable [14, 26]. Vanbekbergen et al.'s SAT formulation of the STG constraints is general in synthesizing an STG. The sizes of constraint formulas directly generated from the STG constraints are usually very large. In practice, it is much easier to satisfy several smaller boolean formulas rather than a single large one. Therefore, we propose a divide-and-conquer approach to handle this problem. Our approach partitions a large state graph into several smaller state graphs has several unique advantages:

- It significantly reduces the number of constraints by several orders of magnitude. For example, for STG benchmark *mr0*, the direct constraint formulation requires the solution of a very large boolean formula with 1,208 variables. In comparison, our divide-and-conquer approach requires the only three very small formulas having 32 variables, 56 variables, and 56 variables, respectively.
- It leads to a reduction in the two-level implementation area. This is due to a good starting point for the logic minimizer and a reduced interaction among circuit signals. For example, for STG benchmark *mr0*, Lavagno et al.'s method requires an implementation area of 86 literals. Vanbekbergen et al.'s method takes prohibitively long time and is unable to yield a solution. In comparison, our divide-and-conquer approach requires an implementation area of only 48 literals.
- It simplifies the circuit verification process.

The divide-and-conquer synthesis methodology does not guarantee the minimum number of state signals to satisfy CSC violations. The number of state signals required to synthesize the STG may increase due to the CSC constraints satisfaction at the decomposed state graph level. However, in practice, we approached global optimum in all the asynchronous benchmark examples, except one, i.e., *mr0*. Even in the case where our solution demands more state signals, e.g., *mr0*, the two-level implementation area of the synthesized circuit is less than the area required to implement the state graph with minimum state signals.

In our approach, initially, we derive a state graph $\sum$ from the given STG specifications. The original state graph specifications are then decomposed into a number of smaller interacting state graphs, one for each primary output o_i. The synthesis process can be summarized as follows :

- Determine the input signal set of an output o_i by greedily removing signals to decrease the CSC violations. Similarly, greedily remove the state signals.
- Derive the projection of the state graph specifications on this input signal set, i.e., derive the decomposed state graph $\sum_{o_i}$ that specifies the functionality of the output signal o_i.
- Find the state signals and their assignment (0, 1, *Up*, *Down*) for the states of SG $\sum_{o_i}$ by finding a truth assignment for the constraint formula representing consistent assignment, semi-modularity, and CSC constraints.
- Propagate the state signal assignments of the new state signals from SG $\sum_{o_i}$ to the states of SG $\sum$.
- These synthesis steps are repeated for every output signal in the SG $\sum$.

The following discussion gives further detail of the divide-and-conquer approach.

3.1 Determining the input signal Set

The *input signal set* belonging to output o_i is defined as the minimum number of STG signals required to implement its logic circuit. The input signal set consists of an immediate input set and some other STG signals required to satisfy CSC constraints. The immediate input set of output o_i is determined by STG signals whose transitions immediately precede a positive or negative transition of output o_i. A signal s_i is in the immediate input set, if and only if, the STG specifies a direct causal relationship between transitions s_i* and o_i*. To minimize the CSC conflicts, the remaining signals in the input set are determined by greedily removing STG signals from the complete state graph $\sum$. A signal s_i that is not in the immediate input set of output o_i can be removed from the state graph if it does not increase the number of CSC conflicts and if it does not increase the state signals required to resolve these conflicts. A signal is removed from the state graph by labeling all its transitions as the silent transition, i.e., ϵ. The removal of the STG signal implies that it is not

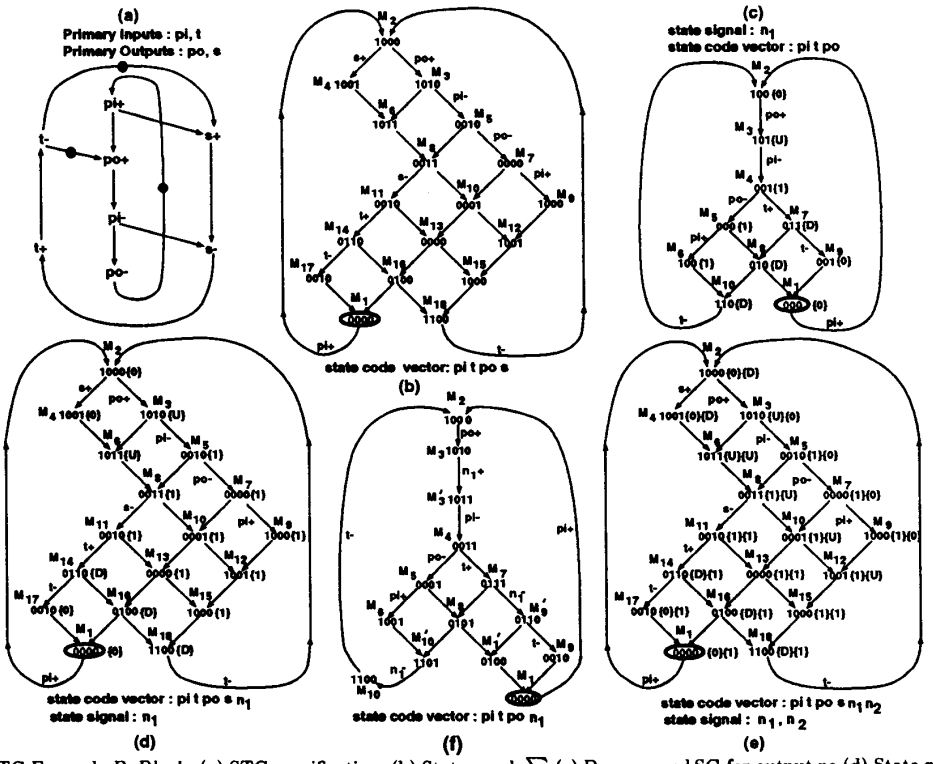

Figure 1: STG Example *PaBlock*: (a) STG specifications (b) State graph $\sum$ (c) Decomposed SG for output *po* (d) State graph $\sum$ with propagated state signal assignment n_1 (e) Decomposed SG for output *s* (in this particular case, this is also the final state graph $\sum$ with CSC) (f) Expanded SG for output *po*.

required to implement the logic circuit corresponding to output o_i. A signal s_i cannot be removed from the state graph if a state signal n_k assigns an *Up* value to state M_i and a *Down* value to state M_j during the transition $M_i \xrightarrow{s_i*} M_j$, or vice-versa. This ensures that the decomposed state graph has a well-defined assignment of state signal n_k. The derivation of state variable assignment in the decomposed state graph is described in Section 3.2.

Example : The example STG in Figure 1(a) has two primary outputs *po* and *s*, and two primary inputs *pi* and *t*. The state code of the state graph is $< pi, t, po, s >$. In the signal transition graph, two signal transitions $t-$ and $pi-$ immediately precede the output signal transitions $po+$ and $po-$, denoted as $t- \rightarrow po+$ and $pi- \rightarrow po-$. Thus, the immediate input set of output *po* consists of two signals, i.e., *pi* and *t*. To find rest of the STG signals in the input signal set, we start from the state graph in Figure 1(b) containing 8 CSC conflicts (see example in Section 2.1). The lower bound on the number of state signals is 2. The STG signal *s* ($\neq$ output *po*), that is not in the immediate input set $\{pi, t\}$, is removed from the state graph (Figure 1(c)). This reduces the number of CSC conflicts to 3 and reduces the lower bound on the number of state signals to 1. No other STG signal ($\neq$ output *po*) can be removed, since they are in the immediate input set $\{pi, t\}$ of output *po*. There are no state signals in the state graph in Figure 1(b). Thus, the input signal set of output *po* consists of signals $\{pi, t, po\}$. Similarly, the input signal set of output *s* is $\{pi, t, po, s, n_1\}$, where n_1 is the state signal.

3.2 Dividing the State Graph

The input signal set belonging to an output is used to derive a decomposed state graph. The decomposed state graph is generated by labeling all the transitions of input set signals as ϵ transitions in the complete state graph $\sum$. The values of the input set signals are removed from the state codes in the complete state graph. Then, the states connected by ϵ transitions are merged together. For example, state M_i and state M_j in the complete state graph transition, $M_i \xrightarrow{\epsilon} M_j$, can be merged into a single state M_iM_j. Removal of ϵ transitions from the state graph is similar to the conversion of a finite automata with ϵ transitions to a finite automata without ϵ transitions [13].

The assignment values of every state signal in the input signal set are propagated to the decomposed state graph as follows:

Consider the state assignment of states M_i and M_j in the complete state graph transition $M_i\{n_{i,1}\}\{n_{i,k}\}\ldots\{n_{i,N}\} \xrightarrow{\epsilon} M_j\{n_{j,1}\}\{n_{i,k}\}\ldots\{n_{i,N}\}$.

Case 1 : If states M_i and M_j have the same assignment value for state signal n_k, i.e., $n_{i,k} = n_{j,k}$, then the merged state M_iM_j in the decomposed state graph will have the same assignment value for state signal n_k. This is illustrated in Figures 2(a), (b), (c), and (d).

Case 2 : If state M_i has the assignment value $n_{i,k} = 0$ and state M_j has the assignment value $n_{j,k} = Up$ (Figure 2(f)), then the transition $M_i\{n_{i,k} = 0\} \xrightarrow{\epsilon} M_j\{n_{j,k} = Up\}$ can

be expanded into a transition sequence $M_i[0] \xrightarrow{\epsilon} M_j'[0] \xrightarrow{n_k+} M_j''[1]$. This new transition sequence is generated by including state signal n_k into the decomposed state graph and including its value into the state code. This process divides state M_j into two different states, $M_j'[0]$ and $M_j''[1]$. States $M_i[0]$ and $M_j'[0]$ have the same state code and they are related by an ϵ transition. Thus, they can be merged together in the decomposed state graph. Finally, the transition sequence $M_iM_j'[0] \xrightarrow{n_k+} M_j''[1]$ is merged into a single state $M_iM_j'M_j''$ (i.e., M_iM_j) with an *Up* assignment to state signal n_k. Similarly, if states M_i and M_j have assignment values $\{n_{i,k} = Up,\ n_{j,k} = 1\}$ $\{n_{i,k} = Down,\ n_{j,k} = 0\}$, and $\{n_{i,k} = 1,\ n_{j,k} = Down\}$, then the merged state M_iM_j will have an assignment value *Up*, *Down*, and *Down*, respectively. This is illustrated in Figures 2(g), (h), and (i).

Case 3 : The rest of the state signal assignments (Figure 2(j)) are inconsistent with the consistent state assignment constraints. Thus, they cannot be assigned by the SAT algorithm.

The remaining CSC constraints in the decomposed state graph are satisfied by deriving a boolean constraint formula (Section 2.1). This constraint formula can be solved by generating a BDD as described in Section 3.3.

Example : In the previous section, the input signal set of output *po* was derived as {*pi*, *t*, *po*}. Since STG signal *s* does not belong to this input set, in order to derive the decomposed state graph for output *po*, all the transitions of *s* are labeled as ϵ transitions. The value of signal *s* is removed from state code $< pi, t, po, s >$ of the complete state graph. Thus, the state code of the decomposed state graph is $< pi, t, po >$. The state sets $\{M_2, M_4\}$, $\{M_3, M_6\}$, $\{M_5, M_8, M_{11}\}$, $\{M_7, M_{10}, M_{13}\}$, and $\{M_9, M_{12}, M_{15}\}$ of complete state graph (see Figure 1(b)) are merged correspondingly into states M_2, M_3, M_4, M_5, and M_6 of the decomposed state graph (Figure 1(c)). Since, at this stage, there are no state signal assignments associated with the complete state graph, the decomposed state graph does not require the propagation of the state signal assignments. The decomposed state graph for output *po* has 3 CSC conflicts and requires *one* state signal to resolve them. The state assignments for a state signal n_1 is derived using the SAT-CSC model in Section 2.1. The decomposed state graph with state signal n_1 assignments is shown in Figure 1(c). The assignment of state signal n_1 is then communicated with the complete state graph $\sum$ (see Figure 1(d)). The same procedure is applied to generate the decomposed state graph for output *s* (shown in Figure 1(g)) from the complete state graph with state signal n_1 (Figure 1(d)).

3.3 BDD based constraint satisfaction

A state variable is a multi-valued variable, that may be assigned a 0, 1, *Up*, or *Down* value. The boolean encoding of a multi-valued state variable requires $\lceil log_2(4) \rceil = 2$ binary variables. The constraint formula represents consistent state assignment, semi-modularity, and CSC constraints on the binary encoded state variables. Vanbekbergen et al. employed a backtracking approach to find a satisfiable assignment for the boolean constraint formula. This framework yields a single satisfiable assignment for the constraint formula, which is often very inefficient in terms of implementation area. In comparison, we developed a BDD based constraint satisfaction algorithm. It yields all the satisfiable assignments for a given constraint formula.

A binary decision diagram (BDD) is a directed acyclic

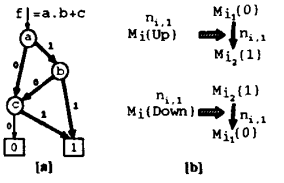

Figure 3: (a) A simple BDD example. (b) *Up* or *Down* state variable assignment.

graph (DAG). The graph has two leaf nodes labeled 0 and 1 representing the boolean functions 0 and 1. Each non-leaf node is labeled with a boolean variable *v* and has two outgoing edges labeled 0 (the left edge) and 1 (the right edge). Every path in a BDD is unique, i.e., no path contains nodes with the same variables. This means that if we arbitrarily trace out a path from the function node to the leaf node 1, then we have automatically found a value assignment to function variables for which function will be 1 regardless of the values of the other variables. Figure 3(a) shows the BDD of a simple function $f = a.b + c$, where all the paths from the root function node to the leaf node 1 are highlighted. Each path yields a satisfiable assignment for the formula *f*.

We employ the algorithms and data structures in [1] to generate a BDD from the boolean constraint formula. The solution to the constraint formula provided by the BDD gives a value assignment to function variables for every state in the decomposed state graph required to satisfy the CSC violations. We can enumerate all the possible assignments to function variables by a simple depth-first traversal of paths in the BDD graph. It is clear that the number of satisfiable assignment (i.e., all the paths from the root node to the leaf 1 node) may be very large for large BDDs. Thus enumeration of all the possible paths may be computationally very expensive. It is known that several logic minimization techniques utilize the number of don't care assignments to yield a reduced circuit implementation area [19]. Since our aim is to minimize the asynchronous circuit implementation area, we chose an assignment (i.e., a path) that maximizes the assignment of don't cares to the state variables. This can be accomplished as follows.

3.4 Implementation Area Reduction using Don't Cares

To minimize implementation area, we must maximize the number of entries in the DON'T CARE sets of functions implementing STG output signals. The network function derivation process (Section 3.6) expands every state with an *Up* or *Down* state variable assignment into two states, one with a 0 assignment and the other with a 1 assignment (as shown in Figure 3(b)). This implies that an *Up* or a *Down* state variable assignment generates two entries in output function ON/OFF sets. On the other hand a 1 or a 0 value assignment to the state variable generates only a single entry in the output function ON/OFF set. Thus the number of entries in the output function's DON'T CARE set can be maximized by minimizing the *Up* or *Down* state variable assignments. In the constraint BDD, a smaller path from the root node to the leaf node 1 assigns 0 or 1 values to lesser number of state variables as compared to number of assignments on a longer path, i.e., the number of entries in the DON'T CARE set can be maximized by choosing a smaller path. This can be accomplished by selecting the shortest path from the root node to the leaf node 1, i.e., by choosing a path that assigns 0 or 1

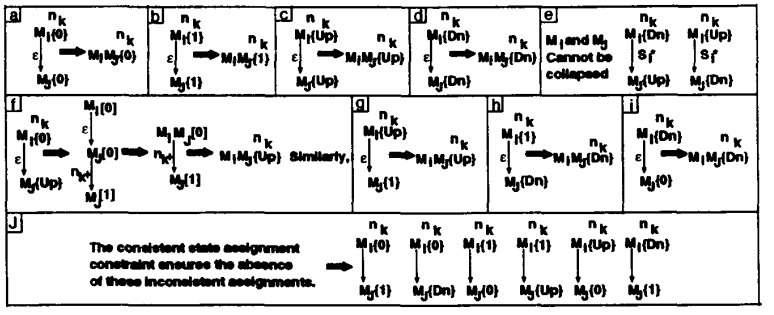

Figure 2: Derivation of state signal (n_k) assignment in the decomposed SG.

```
Input : boolean constraint formula.
Output: A satisfiable assignment with maximum number
    of don't care state variable assignments.
Procedure bdd_bfs_shortest_path( ) {
  Derive a BDD from the given boolean constraint formula ;
  Initialize labeling index i with 0 ;
  Label the BDD root node with 0 ;
  while(leaf node 1 is unlabeled) {
    S_i:={all unlabeled nodes adjacent to at least one node labeled i};
    if(S = {}) stop; /* error: no path from root to leaf node 1 */
    Label all the nodes in S with i + 1 ;
    Increment i ;
  }
  i_l := label of leaf node 1 ;
  Trace the path p from leaf node 1 to the root node
    by choosing successively adjacent nodes with labels
    i_l - 1, i_l - 2, ..., 0 respectively ;
  Output the state variable assignment corresponding
    to the shortest path p ;
}
```

Figure 4: A BDD algorithm for Implementation Area Reduction

values to the minimum number of state variables. The sparse nature of a BDD can be efficiently utilized by employing a simple *breadth first search* (BFS) shortest path algorithm (Figure 4). This algorithm yields a satisfiable assignment with maximum number of don't cares in time complexity[3] $O(n)$, where n denotes the number of BDD nodes. These don't care assignments are exploited by a logic minimizer to yield a substantial reduction in the implementation area.

The solution of the boolean constraint formula gives the number of new state signals required to resolve the CSC violations and the state signal assignment values for every state in the decomposed SG $\sum$. The new assignments resolve all the CSC conflicts in the decomposed state graph. These assignments from the decomposed state graph are then communicated with the complete state graph $\sum$. The propagation of new state signal assignments from the decomposed state graph to the complete state graph is described in the following section.

3.5 State signal assignment propagation

In order to reduce CSC conflicts in the state graph $\sum$, the new state signals in the decomposed graph must be propagated through state graph $\sum$. The process of CSC constraint satisfaction in the decomposed graph and the state assignment propagation to the complete state graph is repeated for every output. Therefore, all the CSC conflicts can be removed from complete state graph $\sum$. In the worst case, all the CSC conflicts in the complete state graph will be removed after all the decomposed state graphs for the output signals are derived.

Definition : A set of states $\{M_1, M_2, \ldots, M_k\}$ in $\sum$ *covers* a state M in a decomposed state graph if states $M_1, M_2, \ldots, M_k$ can be merged with M in the decomposed state graph generation process. This is denoted by $cover(M_1) = cover(M_2) = \ldots = cover(M_k) = M$.

The state assignments are simply propagated by adding the new state signal assignments of state M to states $M_1, M_2, \ldots, M_k$ that cover state M.

Example : For *PaBlock* example, the state signal n_1 assignments are propagated from the decomposed state graph of output *po* (Figure 1(c)) to the complete state graph (Figure 1(d)). In the decomposed state graph generation process (Section 3.2), state sets $\{M_1\}$, $\{M_2, M_4\}$, $\{M_3, M_6\}$, $\{M_5, M_8, M_{11}\}$, $\{M_7, M_{10}, M_{13}\}$, $\{M_9, M_{12}, M_{15}\}$, $\{M_{14}\}$, $\{M_{16}\}$, $\{M_{17}\}$, and $\{M_{18}\}$ in the complete state graph (Figure 1(b)) cover states M_1, M_2, M_3, M_4, M_5, M_6, M_7, M_8, M_9, and M_{10} in the decomposed state graph (Figure 1(c)). Thus, the new state signal n_1's assignment that corresponds to the state M_1 (i.e., 0) in the decomposed state graph (Figure 1(c)) is added to state M_1 in the complete state graph (Figure 1(d)). An assignment corresponding to state M_2 (i.e., 0) in decomposed state graph is added to states M_2 and M_3 in the complete state graph. Similarly, the rest of the state assignments are also propagated to the complete state graph in Figure 1(d). The complete state graph with the new state signal n_1's assignments is shown in Figure 1(d).

3.6 Logic Function Derivation

The partitioning process to generate decomposed state graphs from the complete state graph $\sum$ with new state signal assignments is repeated for every output in the STG. The decomposed state graph for every output can be expanded by a simple procedure to include the state signal transitions into the state graph [26]. The expanded state graph for output *po* is shown in Figure 1(f). Alternatively, we can expand the complete state graph $\sum$ with the state signal assignments (see Figure 1(g)) and derive the output logic functions. The logic function of an output, which is in the sum-of-products form, can then be obtained by simply finding the implied values of the STG outputs in every state of the expanded state graph [3]. A prime-irredundant cover of the output logic function can be obtained by employing a standard logic minimizer, e.g., *espresso*. This cover may contain static and dynamic hazards which can be removed by using some known hazard removal techniques [19].

Figure 5 gives the pseudo-code of our divide-and-conquer synthesis algorithm.

[3]The time complexity of a breadth first search shortest path algorithm is $O(|E|)$, where $|E|$ denotes the number of edges in the graph [5]. A BDD with n nodes has $2n$ edges. Thus the time complexity of a BFS shortest path algorithm on a BDD is $O(n)$

```
Input : STG specifications.
Output: Synthesized logic circuit.

Procedure divide_and_conquer_synthesis(STG) {
  Derive complete state graph ∑ from the given STG;
  Initialize the state signals in the complete state graph ∑ := { };
  for each output signal o_i {
    /*determine input signal set I_S(o_i) belonging to output o_i*/
    state graph ∑_temp := state graph ∑;
    Determine number of CSC conflicts N_csc in ∑_temp;
    Determine lower bound L_b on no. of state signals in ∑_temp;
    Derive the immediate input set I of output signal o_i;
    Initialize input signal set I_S(o_i) := immediate input set I;
    for each signal s_i not in the immediate input set I ∪ o_i {
      Merge states connected with ε or s_i transitions in ∑_temp;
      Determine new CSC conflicts N_csc(new) and
        new lower bound L_b(new);
      if (N_csc(new) <= N_csc and L_b(new) <= L_b) {
        /*signal s_i is not required for logic of output o_i*/
        N_csc := N_csc(new); L_b := L_b(new);
        Label all the transitions of signal s_i in ∑_temp as ε ;
      }
      else I_S(o_i) := I_S(o_i) ∪ s_i;
    }
    for each state signal n_i in ∑_temp
      if (removing n_i increases CSC conflicts in ∑_temp)
        I_S(o_i) := I_S(o_i) ∪ n_i;
    /* derive decomposed state graph ∑_o_i and find new state */
    /* signals, n_s(new), and their assignments in ∑_o_i      */
    Merge states in SG ∑ connected with I_S(o_i) transitions;
    Derive decomposed state graph ∑_o_i;
    for each state signals in I_S(o_i)
      Derive assignment for states in ∑_o_i from states in ∑;
    new state signals n_s(new) := lower bound on state signals
      to resolve CSC conflicts;
    while (no truth assignment found) {
      Derive a boolean constraint formula from ∑_o_i
        with n_s new state signals;
      Find truth assignment for the n_s new state signals
        using the BDD constraint satisfaction algorithm;
      if (the BDD has only tow leaf nodes)
        /*i.e., the boolean formula is unsatisfiable */
        Add a new state signals to n_s(new);
    }
    /*propagate assignment of new state signals from ∑_o_i to ∑*/
    for each state M_k of state graph ∑ {
      if (M_k can be merged in state M'_l of decomposed SG ∑_o_i)
        cover(M_k) := state M'_l of decomposed state graph ∑_o_i;
    }
    for each state M_k of complete state graph ∑
      Add new state assignments n_s(new) from state
        "cover(M_k)" of ∑_o_i to state M_k;
  }
  Expand state graph ∑ to include state signal transitions;
  Derive the logic circuit from the expanded state graph;
  return logic circuit;
}
```

Figure 5: A Divide-and-Conquer Synthesis Algorithm for STGs.

4 Experimental Results

The divide-and-conquer synthesis algorithm was implemented in C language. All the experiments were performed on a SUN-SPARC 2 workstation. We tested our algorithm on a large number of STG benchmarks including the HP-benchmarks [14]. We also compared the performance of our algorithm with other well known techniques, e.g., Lavagno et al. [14] and Vanbekbergen et al.'s [4] technique [26]. The results of these experiments are given in Table 1. The results indicate that our algorithm outperforms both Lavagno et. al.'s [14] and Vanbekbergen et al.'s [26] algorithm in terms of execution time. For example, in the case of a large STG example, i.e., *mr0*, our algorithm requires 58.36 seconds to yield a two-level implementation area of 48 literals, as compared to 1084.5 seconds required for an implementation area of 86 literals with Lavagno et al.'s algorithm. For this example, Vanbekbergen et al.'s algorithm requires the solution of a large constraint formula with 1,208 variables. In comparison, our divide-and-conquer approach requires the solution of only three very small constraint formulas, one with 32 variables and the other two with 56 variables each. These formulas can be solved in a very short execution time of 58.36 CPU seconds, as compared to the SAT backtrack limit failure in > 3600 seconds for the brute force approach. In practice, it is difficult for Vanbekbergen et al.'s technique [26] to synthesize interface circuits having nore than 100 STG states.

We also calculated the 2-level area of the synthesized logic by finding a prime-irredundant cover from logic minimizer *espresso* (single output exact minimization option, i.e., *espresso* -Dso -S1). The two-level area results with Lavagno et al.'s were also calculated from the prime-irredundant cover of the network logic function (option **astg_syn -r** in U.C.Berkeley logic synthesis tool SIS). Our algorithm yields a reduced two-level implementation area (Table 1) than both the Lavagno et al.'s and Vanbekbergen et al.'s algorithms for all the circuits in the asynchronous benchmark set [14]. Lavagno et al.'s method yields a total area of 534 literals in 1315.3 seconds. In comparison, for the same benchmarks, our approach achieves an area of 447 literals in 322.25 seconds. Similarly, Vanbekbergen et al.'s method yields an area of 500+ literals in (6.05 + 101.3 + >3600) seconds and it is unable to solve large examples such as *mr0* and *mmu1*. For the same benchmarks, Our divide-and-conquer approach yields a total area of 399 literals in 246.18 seconds. These results show that compared to the existing techniques, the divide-and-conquer technique is capable of achieving an average of 20% reduction in implementation area for all the benchmarks and it offers a practical solution for the complex asynchronous interface design problems.

[4] Our implementation of Vanbekbergen et al.'s synthesis algorithm gives much better runtime results than those reported by the authors in [26].

5 Conclusion

An efficient divide-and-conquer synthesis approach for asynchronous interface circuit design is presented. This approach partitions a large STG into smaller and manageable modules that significantly reduces the number of design constraints. In practice, signal transition graphs with large number of states can be synthesized efficiently. Experimental results with a large number of practical STG benchmarks indicate that, compared to the existing techniques, this divide-and-conquer approach produces a circuit design with reduced area and achieves significant performance improvement in terms of computing time and implementation area. It offers a practical synthesis tool to solve complex asynchronous circuit design problems.

Acknowledgments

We would like to thank Luciano Lavagno for providing us with their STG state assignment program [14]. We also thank Ganesh Gopalakrishnan, Steve Nowick, and Peter Vanbekbergen for helpful discussions.

Table 1: Experimental results with practical STG benchmarks on a SUN SPARC-2 workstation.

STG Name	Specifications		BDD Method				Vanbekbergen et al.				Lavagno and Moon			
	Initial no. of states	Initial no. of signal	Final no. of states	Final no. of signal	2level Area literals†	CPU time sec.	Final no. of states	Final no. of signal	2level Area literals†	CPU time sec.	Final no. of signal	2level Area literals†	CPU time sec.
mr0	302	11	884	15	48	58.36	SAT Backtrack Limit			> 3600	13	86	1084.5
mmu1	82	8	111	10	38	28.16	SAT Backtrack Limit			101.3	10	37	47.8
sbuf-ram-write	58	10	99	12	47	32.79	90	12	74	5.21	12	35	54.6
vbe4a	58	6	106	8	30	1.95	116	8	40	0.25	8	41	5.5
nak-pa	56	9	59	10	25	0.53	58	10	32	0.08	10	41	20.8
pe-rcv-ifc-fc	46	8	50	9	49	85.55	53	9	50	0.13	9	62	14.8
ram-read-sbuf	36	10	40	11	25	0.25	53	11	44	0.06	11	28	65.2
alex-nonfc	24	6	28	7	21	0.87	28	7	22	0.03	Non-Free-Choice STG		
sbuf-send-pkt2	21	6	23	7	17	0.87	27	7	29	0.04	7	14	8.6
sbuf-send-ctl	20	6	28	8	30	18.27	28	8	35	0.03	8	48	3.4
atod	20	6	25	7	14	0.15	24	7	16	0.01	7	19	2.9
pa	18	4	32	6	17	10.08	31	6	22	0.06	Internal State Error*		
alloc-outbound	17	7	24	10	21	91.43	24	9	27	0.04	9	28	2.5
wrdata	16	4	20	5	18	0.05	19	5	18	0.01	5	21	0.9
fifo	16	4	23	5	15	0.14	20	5	17	0.02	5	15	0.7
sbuf-read-ctl	14	6	18	7	16	0.05	16	7	20	0.01	7	15	1.5
nousc	12	5	16	4	12	0.09	16	4	12	0.01	4	14	0.5
vbe-ex2	8	2	12	4	18	3.94	12	4	18	0.03	4	21	0.5
nousc-ser	8	3	10	4	9	0.06	10	4	9	0.01	4	11	0.4
sendr-done	7	3	9	4	8	0.05	10	4	8	0.01	4	6	0.4
vbe-ex1	5	2	7	3	7	0.03	8	3	7	0.01	3	7	0.3

† The number of literals were calculated from the unfactored prime-irredundant cover obtained using espresso -Dso -S1 options.

* The state splitting technique of [14] has not been implemented in the Berkeley tool SIS, which results in internal state error in some cases.

References

[1] K. S. Brace, R. L. Rudell, and R. Bryant. Efficient Implementation of a BDD package. In *Proc. of 27th DAC*, pages 40–45, 1990.

[2] R. E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Trans. on Computers*, 35(8):677–691, August 1986.

[3] Tam-Anh Chu. *Synthesis of Self-Timed VLSI Circuits from Graph-theoretic Specifications*. PhD thesis, Dept. of Electrical Engineering and Computer Science, MIT, June 1987.

[4] S. A. Cook. The Complexity of Theorem-Proving Procedures. In *Proc. of Third Annual ACM Symposium on Theory of Computing*, pages 151–158, 1971.

[5] S. Even. *Graph Algorithms*. Computer Science Press, 1979.

[6] J. Gu. Efficient local search for very large-scale satisfiability problem. *SIGART Bulletin*, 3(1):8–12, Jan. 1992, ACM Press.

[7] J. Gu. The *UniSAT* problem models (appendix). *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 14(8):865, Aug. 1992.

[8] J. Gu. Local search for satisfiability (SAT) problem. *IEEE Trans. on Systems, Man, and Cybernetics*, 23(4):1108–1129, Jul./Aug. 1993.

[9] J. Gu. Global optimization for satisfiability (SAT) problem. *IEEE Trans. on Knowledge and Data Engineering*, 6(3), Jun. 1994.

[10] J. Gu. *Optimization Algorithms for Satisfiability (SAT) Problem. In* New Advances in Optimization and Approximation. *Ding-Zhu Du (ed)*, pages 72–154. Kluwer Academic Publishers, Boston, MA, Jan. 1994.

[11] J. Gu, P.W. Purdom, and B.W. Wah. Algorithms for Satisfiability (SAT) Problem: A Survey. 1994, To appear.

[12] J. Gu and R. Puri. A Preprocessor for Satisfiability Testing: A Case Study in Asynchronous Circuit Synthesis. 1994, To appear.

[13] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computations*. Addison-Wesley Publishing Co., 1987. Chapter 2.

[14] L. Lavagno, C. Moon, R. Brayton, and A. Sangiovanni-Vincentelli. Solving the State Assignment Problem for Signal Transition Graphs. In *Proc. of 29th DAC*, pages 568–572, 1992.

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

[16] C. W. Moon, P. R. Stephan, and R. K. Brayton. Specification, Synthesis and Verification of Hazard-Free Asynchronous Circuits. In *Proc. of ICCAD*, pages 322–325, 1991.

[17] T. Murata. Petri nets: Properties, analysis and applications. *Proc. of the IEEE*, 77(4):541–580, April 1989.

[18] C. Myers and T. H. Meng. Synthesis of Timed Asynchornous Circuits. *IEEE Trans. on VLSI Systems*, 1(2):106–119, June 1993.

[19] S. M. Nowick and D. L. Dill. Exact Two-level Minimization of Hazard-free Logic with Multiple-input Changes. In *Proc. of ICCAD*, 1992.

[20] R. Puri and J. Gu. An Efficient Algorithm for Microword Length Minimization. *Local Search for the Satisfiability Problem* (Appendix). In *Proc. of 29th DAC*, pages 651–656, 1992.

[21] R. Puri and J. Gu. An Efficient Algorithm to Search for Minimal Closed Covers in Sequential Machines. *IEEE Trans. on CAD*, 12(6):737–745, June 1993.

[22] R. Puri and J. Gu. A Modular Partitioning Approach for Asynchronous Circuit Synthesis. In *Proc. of 31st DAC*, 1994.

[23] J. H. Tracey. Internal State Assignment for Asynchronous Sequential Machines. *IEEE Trans. on Computers*, 15(4):551–560, August 1966.

[24] P. Vanbekbergen. Personal communication, Sept., 1993.

[25] P. Vanbekbergen, G. Goossens, F. Catthoor, and H. De Man. Optimized Synthesis of Asynchronous Control Circuits from Graph-Theoretic Specifications. *IEEE Trans. on CAD*, 11(11):1426–1438, Nov. 1992.

[26] P. Vanbekbergen, B. Lin, G. Goossens, and H. De Man. A Generalized State Assignment Theory for Transformations on Signal Transition Graphs. In *Proc. of ICCAD*, pages 112–117, 1992.

[27] M. L. Yu and P. A. Subrahmanyam. A New Approach for Checking the Unique State Coding Property of Signal Transition Graphs. In *Proc. of EDAC*, pages 312–321, 1992.