A Modular Partitioning Approach for Asynchronous Circuit Synthesis[†]

Ruchir Puri and Jun Gu
Dept. of Electrical & Computer Engineering
University of Calgary, Calgary, Canada T2N 1N4
{puri,gu}@vlsi.enel.ucalgary.ca

Abstract

Asynchronous circuits are crucial in designing *low power* and *high performance* digital systems. In this paper, we present an efficient modular partitioning approach for asynchronous circuit synthesis. This approach partitions a large circuit specification into smaller and manageable modules that drastically reduces the synthesis complexity. Experimental results with a large number of practical asynchronous benchmarks are presented. They show that, compared to the existing techniques, this modular partitioning method achieves many orders of magnitude of performance improvement in terms of computing time, in addition to a reduced implementation area. It offers a practical solution for complex asynchronous circuit design problems.

1 Introduction

Asynchronous interface circuits are indispensable in many real-time digital systems. Due to potential applications in low power and speed-independent systems, recently, there has been a renewed interest in the automated synthesis of asynchronous interface circuits [1, 12, 14, 21]. Previous researchers have developed an event-based graphical specification [1] and a direct synthesis method [1, 12, 21, 23]. They are unable to synthesize complex designs involving large number of constraints. The modular partitioning synthesis approach proposed in this paper offers a practical solution to handle complex asynchronous circuit design problems.

The event-based graphical specifications, also called *signal transition graphs* (STG) [1], are based on petri nets. They are very powerful in describing the behavior of asynchronous interface circuits. To implement the asynchronous behavior into a logic circuit, the STG specifications must satisfy complete state coding (CSC) constraints. Informally, the CSC constraint means 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 asynchronous interface circuits do not satisfy the CSC constraints. The original STG must be transformed to satisfy them. Most methods are limited by the type of asynchronous interface behaviors they can synthesize. Lin et al. [14], Vanbekbergen et al. [21], and Yu et al. [23] proposed synthesis techniques that are restricted to the STG specifications describing only concurrent asynchronous behavior (i.e., marked graphs). These techniques were limited by an additional restriction that every signal has exactly one rising and one falling transition. This restricts the synthesis of asynchronous circuits from general STG specifications. Lavagno and Moon et al. [13] proposed a new synthesis framework for STG specifications with a limited interplay of concurrency and choice (i.e., safe free-choice petri 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 with state minimization [17] and critical race-free state assignment techniques. The algorithm inserts state signals into the original STG to satisfy CSC constraints. It handles live-safe free choice nets. Recently, Vanbekbergen et al. [22] proposed a general 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 the SAT problem. For example, a moderately large size signal transition graph with 174 states generates a boolean formula with 35,386 clauses and 1,044 variables. In our experience [2, 3, 4, 6, 7, 9, 8, 16], it usually takes prohibitively long time to find a satisfiable assignment for very large boolean formulas.

In this paper, we propose a modular partitioning approach for the synthesis of asynchronous circuits from signal transition graphs. For a given problem, its STG is first partitioned into a number of simpler and manageable "modules." Each modular graph is then solved individually. Eventually, the results of these small graphs are integrated together, which contribute to the solution of the given problem. This approach of partitioning the

[†]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.

state graph into smaller modules avoids the problem of solving very large SAT formulas. The modular partitioning approach is capable of synthesizing asynchronous circuits from general signal transition graph specifications. It is not limited to marked graph or safe free-choice petri nets. Compared to Lavagno et. al.'s [13] and Vanbekbergen et al.'s [22] algorithms, this approach achieves many orders of magnitude of performance improvement in terms of computing time, in addition to a reduced implementation area.

The rest of this paper is organized as follows. In Section 2, we give some basic definitions and notations that simplify our discussion. In Section 3, we describe the modular partitioning synthesis approach in detail. Experimental results with practical asynchronous STG benchmarks and performance comparisons with existing methods are given in Section 4. Section 5 concludes this paper.

2 Preliminaries

A *petri net* [15] 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* $F \subseteq P \times \mathrm{T} \cup \mathrm{T} \times \mathrm{P}$ (represented as directed arcs) specifying a binary relation between transitions and places. The dynamic behavior is captured by the petri net *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 the 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 circuits. Transitions s_i+, s_i-, and s_i* denote a rising, a falling, and a rising or falling transitions on signal wire s_i respectively. 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}. 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 [1]. The state graph is a finite automaton that represents 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 [15]. A state graph can be mapped into a circuit by satisfying the 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.

The CSC constraint is the necessary and sufficient constraint to derive the logic circuit functions from the state graph [18]. A CSC violation must be corrected by inserting new signals in the state graph, so as to distinguish between the states violating CSC constraint [13, 22]. 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. [22]. They formulated the CSC problem as a boolean satisfiability (SAT) problem.

In the following section, we briefly describe a SAT formulation of the CSC constraint satisfaction problem.

2.1 A SAT model for CSC satisfaction

The satisfiability (SAT) model for CSC satisfaction, 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 which are 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 have the same state encoding. The complexity of the extracted boolean formula depends on the number of states N, the number of concurrent transitions, and the number of CSC constraints. For a state graph with N states and E edges, the boolean formula will have $m{\times}(c_1.E + c_2.N_{ct} + N_{usc}.{c_3}^m + N_{csc}.{c_4}^m)$ clauses and $2 \times N \times m$ variables where m is the number of state signals required to satisfy the CSC constraints, N_{ct} is the number of concurrent transitions, N_{usc} is the number of state pairs that have the same binary encoding, N_{csc} is the number of CSC constraints, and c_1, c_2, c_3, and c_4 are constants.

The SAT 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 SAT formula. The *goal* is to find a truth assignment to the state variables so that all the constraints are satisfied.

The following section describes our modular partitioning approach to asynchronous circuit synthesis.

3 A Modular Partitioning Approach for Constraint Satisfaction

Most techniques proposed for the synthesis of asynchronous circuits from signal transition graphs are restricted in practical applications [14, 21, 23]. Some of them try to satisfy all the constraints in the state graph directly, which, in most cases, is clearly intractable [13, 22]. Vanbekbergen et al.'s SAT formulation of the STG constraints is general in synthesizing an STG. The sizes of SAT 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. It is natural to use a modular partitioning approach to handle this problem.

3.1 A model for modular partitioning

A constraint satisfaction problem has three components: variables, values, and constraints. The *goal* is to find an assignment of values to variables such that all the constraints are satisfied [5].

In a *modular* constraint graph model, the complete graph is partitioned into smaller and simpler graphs [5]. Various local, smaller graphs can be manipulated individually. An integration mechanism is used that fits these *local* modules together into a *global* network. Modularity refers to the ability of a constraint satisfaction model to decompose a complex system into easily understood modules.

[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}, 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.

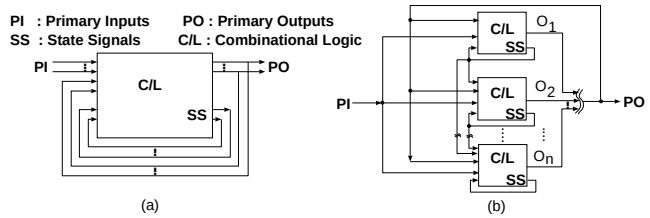

Figure 1: Asynchronous circuit synthesis (a) Direct approach. (b) A modular partitioning approach.

An important aspect of modularity is constraints [5]. Modularity of constraints refers to the ability of a constraint satisfaction model to enable complex information to be represented in terms of modules of local information. Each individual module of constraint relations may be handled separately. However, the constraint satisfaction representations differ with respect to how a module of local information *communicates* with another. In our model, the assignment information in each local module is communicated to other local graphs as well as with the global graph. It is this interaction which allows the complete solution to be built from the solutions of the individual local modules.

In this modular partitioning approach, for a given problem, the state graph is first partitioned into a number of simpler and manageable state graphs. Each modular graph is then solved individually. The results of these small graphs are integrated together, which contribute to the solution of the given large problem.

The partitioning of a large state graph into smaller state graphs has several unique advantages:

- It significantly reduces the number of constraints by several orders of magnitude. For example, for STG benchmark *mmu*0, the direct SAT formulation requires the solution of a very large SAT formula with 35,386 clauses. In comparison, our modular partitioning approach requires the only three very small formulas having 954 clauses, 954 clauses, and 85 clauses.
- 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.
- It simplifies the circuit verification process.

The modular partitioning approach is illustrated in Figure 1. It consists of the following several steps :

- Determine the input signal set (denoted as $I_S(o_i)$), belonging to output o_i by greedily removing signals from complete graph (denoted as $\sum$) to decrease the CSC conflicts. Similarly, greedily remove the state signals.
- Derive a smaller modular state graph (denoted as $\sum_{o_i}$) from the complete graph $\sum$, for the input set, $I_S(o_i)$.
- Find the new state signals and their assignments (0, 1, Up, $Down$) to the states of graph $\sum_{o_i}$ by finding a truth assignment to the SAT formula representing the

```
procedure determine_input_set(∑, o_i, I_S(o_i)) {
  state graph ∑_temp := state graph ∑;
  determine number of CSC conflicts N_csc in ∑_temp;
  determine the lower bound L_b on number 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 immediate input set I ∪ o_i {
     merge states connected with ϵ or s_i
        transitions in ∑_temp;
     determine new CSC conflicts N_csc(new);
     determine 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 function 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 transitions;
     }
     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;
}
```

Figure 2: An algorithm for input set derivation.

consistent assignment, semi-modularity, and CSC constraints.

- Propagate the truth assignments to the new state signals from graph $\sum_{o_i}$ to the complete graph $\sum$.
- Repeat the above steps for every output signal.

The above procedure yields a modular topology as shown in Figure 1(b). It is equivalent to the complete state graph $\sum$.

This modular synthesis method does not guarantee an optimal solution, i.e., it does not guarantee the minimum number of state signals to satisfy CSC constraints. The number of state signals required to synthesize the STG may increase due to the integration of local solutions from the modular state graphs. However, in practice, we have achieved global optimum in all other asynchronous benchmarks, except two. Although for STG $mr0$, our solution requires more state signals, 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 the following, we discuss the major steps in our modular partitioning approach in detail.

3.2 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

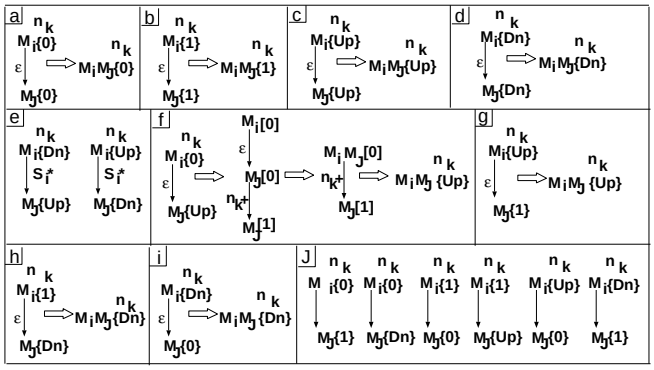

Figure 3: The derivation of state signal assignments in the modular state graph.

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 ($\neq$ output o_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 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 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 in the transition $M_i \xrightarrow{s_i*} M_j$, or vice-versa. This ensures that the modular state graph has a well-defined assignment of state signal n_k. The derivation of state variable assignment in the modular state graph is described in Section 3.3.

The above procedure to determine the input set is summarized in algorithm *determine_input_set()* in Figure 2.

3.3 Modular state graph generation and constraint satisfaction

The input signal set belonging to an output is used to derive a modular state graph. The modular 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

```
procedure partition_sat(∑, ∑_{o_i}, n_s(new)) {
  merge states in state graph ∑ connected
     with I_S(o_i) transitions;
  derive modular state graph ∑_{o_i};
  for each state signals in I_S(o_i)
     derive the assignment for states in ∑_{o_i} from
        the 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;
     if (the boolean formula is unsatisfiable)
        add a new state signals to n_s(new);
  }
}
```

Figure 4: An algorithm for modular state graph generation and constraint satisfaction.

[10].

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

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

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 modular state graph will also have the same assignment value for state signal n_k. This is illustrated in Figures 3(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 3(f)), then the transition $M_i\{n_{i,k} = 0\} \stackrel{\epsilon}{\rightarrow} M_j\{n_{j,k} = Up\}$ can be expanded into a transition sequence $M_i[0] \stackrel{\epsilon}{\rightarrow} M_j'[0] \stackrel{n_k+}{\rightarrow} M_j''[1]$. This new transition sequence is generated by including state signal n_k into the modular 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 modular state graph. Finally, the transition sequence $M_iM_j'[0] \stackrel{n_k+}{\rightarrow} M_j''[1]$ 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 3(g), (h), and (i).

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

```
procedure propagate(∑_{o_i}, ∑, n_s(new)) {
  for each state M_k of state graph ∑ {
     if (state M_k can be merged in state M_l' of
        modular state graph ∑_{o_i})
           cover(M_k) := state M_l' of modular
              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;
}
```

Figure 5: An algorithm for assignment propagation.

SAT algorithm. The remaining CSC constraints in the modular state graph are satisfied by deriving a boolean satisfiability formula (Section 2.1). The solution of this SAT formula gives the state signal assignment values for every state in modular state graph. The new assignments resolve all the CSC conflicts in the modular state graph. These assignments from the modular state graph are then communicated with the complete state graph $\sum$.

The above procedure to generate the modular state graph and to satisfy STG constraints is summarized in the algorithm *partition_sat()* in Figure 4. The propagation of new state signal assignments from the modular state graph to the complete state graph is described in the following section.

3.4 Propagation of state signal assignment

In order to reduce CSC conflicts in the state graph $\sum$, the new state signals in the modular graph must be propagated through state graph $\sum$. The process of CSC constraint satisfaction in the modular 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 modular 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 modular state graph if states $M_1, M_2, \ldots, M_k$ can be merged into M in the modular 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.

The above procedure to propagate new state signal assignments from modular state graph to complete state graph $\sum$ is summarized in algorithm *propagate()* in Figure 5.

3.5 Logic function derivation

The partitioning process to generate modular state graphs from the complete state graph $\sum$ with new state signal assignments is repeated for every output in the

```
procedure modular_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 the input signal set I_S(o_i)
        belonging to output o_i */
     determine_input_set(∑, o_i, I_S(o_i));
     /* derive modular state graph ∑_{o_i} and
        find new state signals, n_s(new), and
        their assignments in ∑_{o_i} */
     partition_sat(∑, ∑_{o_i}, n_s(new));
     /* propagate the assignment of new
        state signals from ∑_{o_i} to ∑ */
     propagate(∑_{o_i}, ∑, n_s(new));
  }
  expand state graph ∑ to include state
     signal transitions;
  derive logic circuit from expanded state graph;
  return logic circuit;
}
```

Figure 6: An algorithm for modular synthesis of asynchronous circuits from STGs.

STG. The modular state graph for every output can then be expanded by a simple procedure to include the state signal transitions into the state graph [22]. Alternatively, we can expand the complete state graph $\sum$ with the state signal assignments 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 [1]. 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 [12].

The complete procedure for modular synthesis of asynchronous circuits from signal transition graphs is summarized in the algorithm *modular_synthesis()* in Figure 6.

4 Experimental Results

The algorithm for modular synthesis of asynchronous circuits was implemented in C language. We employed an efficient implementation of a branch and bound algorithm[3] to solve the SAT formulas [20]. We tested our algorithm on a large number of STG benchmarks including the HP-benchmarks [13]. All the experiments were performed on a SUN-SPARC 2 workstation. We also compared the performance of our algorithm with other well known techniques, e.g., Lavagno et al. [13] and Vanbekbergen et al.'s [22] algorithms. The results of these experiments are given in Table 1. The results indicate that our algorithm outperforms both Lavagno et. al.'s [13] and Vanbekbergen et al.'s [22] algorithms in terms of execution time and implementation area. For example, in the case of a large STG example, i.e., *mr*0, our algorithm requires 2.80 seconds to yield a two-level implementation area of 41 literals, as compared to 1084.5 seconds and an area of 86 literals with Lavagno et al.'s algorithm. For this example, Vanbekbergen et al.'s algorithm cannot yield a solution within 3600 seconds and the program was aborted due to backtracking limit. For another STG benchmark, i.e., *mmu*0, the direct SAT formulation requires the solution of a large SAT formula with 35,386 clauses and 1,044 variables. In comparison, our modular synthesis approach requires the solution of only three very small SAT formulas, one with 85 clauses and 18 variables and the other two with 954 clauses, 96 variables each. These formulas can be solved in 0.87 seconds, as compared to a pre-aborted 406.3 seconds for the brute force approach [22].

We also calculated the two-level area of the logic circuit synthesized by finding a prime-irredundant cover from logic minimizer *espresso*. We ran *espresso* with single output exact minimization option, i.e., *espresso* -Dso -S1. The results of two-level area from Lavagno et al.'s algorithm were also calculated from the prime-irredundant cover of the network logic function. We employed **astg_syn -r** option in the U.C.Berkeley logic synthesis tool SIS to find the prime-irredundant cover. On average, our modular partitioning algorithm reduces the two-level implementation area by 12% than that of the Vanbekbergen's direct synthesis method. As compared to Lavagno et al.'s algorithm, we obtained an average area improvement of 9%. The two-level implementation area results are summarized in Table 1. The implementation area was further reduced by developing a BDD based constraint satisfaction approach [19].

5 Conclusion

An efficient modular partitioning approach for asynchronous circuit design is presented. This approach partitions a large STG into smaller and manageable modules that significantly reduce the design complexity. In practice, signal transition graphs with very large number of states can be synthesized in a very short execution time. Experimental results with a large number of practical STG benchmarks indicate that, compared to existing techniques, this modular partitioning method achieves many orders of magnitude of performance improvement in terms of computing time, in addition to a reduced implementation area. It offers a practical solution to complex asynchronous circuit design problems.

Acknowledgements

We thank Luciano Lavagno for providing us with their STG state assignment program [13]. We also thank Ganesh Gopalakrishnan, Steve Nowick, and Peter Vanbekbergen for helpful discussions.

[3] The SAT program is provided with the U.C.Berkeley logic synthesis tool *SIS*.

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

STG Name	Specifications		Our Method (Decomposition)				Vanbekbergen et al. [22] (No Decomposition)				Lavagno and Moon et al. [13]		
	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	469	14	41	2.80	SAT Backtrack Limit			> 3600	13	86	1084.5
mr1	190	8	373	12	55	1.73	SAT Backtrack Limit			872.9	10	53	237.5
mmu0	174	8	441	11	49	0.87	SAT Backtrack Limit			406.3	Internal State Error*		
mmu1	82	8	131	10	50	0.37	SAT Backtrack Limit			101.3	10	37	47.8
sbuf-ram-write	58	10	93	12	59	0.36	90	12	74	5.21	12	35	54.6
vbe4a	58	6	106	8	37	0.19	116	8	40	0.25	8	41	5.5
nak-pa	56	9	59	10	25	0.20	58	10	32	0.08	10	41	20.8
pe-rcv-ifc-fc	46	8	50	9	48	0.24	53	9	50	0.13	9	62	14.3
ram-read-sbuf	36	10	44	11	28	0.15	53	11	44	0.06	11	23	65.2
alex-nonfc	24	6	31	7	26	0.05	28	7	22	0.03	Non-Free-Choice STG		
sbuf-send-pkt2	21	6	26	7	20	0.04	27	7	29	0.04	7	14	8.6
sbuf-send-ctl	20	6	32	8	33	0.09	28	8	35	0.03	8	43	3.4
atod	20	6	26	7	15	0.02	24	7	16	0.01	7	19	2.9
pa	18	4	34	6	18	0.12	31	6	22	0.06	Internal State Error*		
alloc-outbound	17	7	29	9	33	0.09	24	9	27	0.04	9	23	2.5
wrdata	16	4	20	5	17	0.03	19	5	18	0.01	5	21	0.9
fifo	16	4	23	5	15	0.03	20	5	17	0.02	5	15	0.7
sbuf-read-ctl	14	6	18	7	16	0.06	16	7	20	0.01	7	15	1.5
nousc	12	3	16	4	12	0.01	16	4	12	0.01	4	14	0.5
vbe-ex2	8	2	12	4	18	0.08	12	4	18	0.03	4	21	0.5
nousc-ser	8	3	10	4	9	0.02	10	4	9	0.01	4	11	0.4
sendr-done	7	3	10	4	8	0.02	10	4	8	0.01	4	6	0.4
vbe-ex1	5	2	8	3	7	0.01	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 [13] has not yet been implemented in the U.C.Berkeley logic synthesis tool *SIS*, which results in an *internal state error* in some cases [11].

References

[1] 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.

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

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

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

[5] J. Gu. *Constraint-Based Search*. Cambridge University Press, New York, 1994.

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

[7] 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.

[8] J. Gu, P.W. Purdom, and B.W. Wah. Algorithms for Satisfiability (SAT) Problem: A Survey. 1993, To appear.

[9] J. Gu and R. Puri. A Preprocessor for Satisfiability Testing: A Case Study in Asynchronous Circuit Synthesis. 1993, To appear.

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

[11] L. Lavagno. Personal communication, Sept., 1993.

[12] L. Lavagno, K. Keutzer, and A. Sangiovanni-Vincentelii. Algorithms for Synthesis of Hazard-free Asynchronous Circuits. In *Proc. of 28th DAC*, pages 302–308, 1991.

[13] 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.

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

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

[16] 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.

[17] 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.

[18] R. Puri and J. Gu. Asynchronous Circuit Synthesis: Persistency and Complete State Coding Constraints in Signal Transition Graphs. *International Journal of Electronics*, 75(5):933–940, Nov. 1993.

[19] R. Puri and J. Gu. A Devide and Conquer Approach for Asynchronous Interface Synthesis. In *Proc. of 7th IEEE/ACM International High-Level Synthesis Syposium*, May, 1994.

[20] P. R. Stephan, R. K. Brayton, and A. Sangiovanni-Vincentelli. Combinational Test Generation Using Satisfiability. Technical Report ERL M92/112, U.C.Berkeley, Oct., 1992.

[21] 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.

[22] 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.

[23] 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.