Synthesis of Speed Independent Circuits Based on Decomposition

Tomohiro Yoneda*
National Institute of Informatics
yoneda@nii.ac.jp

Hiroomi Onda
Tokyo Institute of Technology
onda@yt.cs.titech.ac.jp

Chris Myers†
University of Utah
myers@ece.utah.edu

Abstract

This paper presents a decomposition method for speed-independent circuit design that is capable of significantly reducing the cost of synthesis. In particular, this method synthesizes each output individually. It begins by contracting the STG to include only transitions on the output of interest and its trigger signals. Next, the reachable state space for this contracted STG is analyzed to determine a minimal number of additional signals which must be reintroduced into the STG to obtain CSC. The circuit for this output is then synthesized from this STG. Results show that the quality of the circuit implementation is nearly as good as the one found from the full reachable state space, but it can be applied to find circuits for which full state space methods cannot be successfully applied. The proposed method has been implemented as a part of our tool `nutas` *(Nii-Utah Timed Asynchronous circuit Synthesis system), and its very first version is available at* `http://research.nii.ac.jp/~yoneda`.

Key Words: *Decomposition, synthesis, STGs, abstraction, speed-independent circuits.*

1. Introduction

Logic synthesis [1, 2, 3] from low level specification languages is one of the major approaches to the automated synthesis of asynchronous circuits. This approach can potentially synthesize more optimized circuits with higher performance than other methods such as syntax directed translation method [4, 5, 6, 7, 8, 9]. It, however, usually requires an enumeration of the state space of the given specification, and it often suffers from the state explosion problem. Thus, large specifications expressed in hardware description languages have usually been synthesized by syntax directed translation methods or similar techniques that do not require state space enumeration, sometimes with local optimization techniques such as [10]. This paper tackles the challenge of using logic synthesis also for large specifications derived from hardware description languages, as it has the potential of in the future providing further global optimization through timed circuit synthesis [11]. In this approach, a specification written in some high-level language is first translated to a signal transition graph (STG), and, then logic synthesis is applied to this STG. This method requires a compiler to generate STGs with the complete state coding (CSC) property, and an efficient logic synthesis method. A preliminary tool for the former is described in [12], and improved version is described in [13]. Guaranteeing CSC by such a correct-by-construction method, which may not give optimal solutions in the number of inserted state variables, is practical for large STGs, because automatic CSC solvers sometimes do not handle such STGs well. Note that only a small number of inserted state variables are actually used to implement each output, and so, the delays of the circuits are not significantly affected even in a non-optimal solution. This paper is for handling the latter issue, and aims at reducing the average cost for logic synthesis from STGs by decomposing a specification and running the logic synthesis procedure for each small sub-specification.

The idea for decomposition based synthesis is first proposed by Chu [14]. In his work, one primary output is picked up, and the given STG is modified by replacing each transition for the signal that does not affect the output by a dummy transition. Then, the modified STG is reduced by eliminating selected dummy transitions while preserving the behavior. A correct circuit can be synthesized from this reduced STG with usually much smaller cost. This work, however, had two open problems. First, the reduction of STGs, called *contraction*, was not formalized. For a simple STG such as a marked graph, its contraction is easy. But, in the general case, the formalized algorithm was unknown at that time. Second, it was not straightforward to decide if a signal actually affects the output signal or not, and no algorithm to make this decision is given in his thesis. As for the first problem, Vogler and Wollowski recently formalized the contraction algorithm using a bisimulation relation in [15], and Zheng and Myers developed a timed contraction algorithm in [16]. On the other hand, Puri and Gu tried to solve the second problem in [17]. Their algorithm greedily removes an irrelevant signal (with respect to the output signal) such that the number of CSC conflicts does not increase by hiding the signal. This algorithm is, however, not

* This research is supported by JSPS Joint Research Projects.
† This research is supported by NSF Japan Program award INT-0087281 and SRC grant 2002-TJ-1024.

so helpful for our purpose, because it needs the state graph of the original STG, which cannot be constructed due to state explosion for very large STGs. Beister, Eckstein, and Wollowski proposed a similar decomposition based method for extended-burst-mode machines [18].

The main contribution of this work is to propose a new algorithm to find a sufficient set of input signals for a given output for the decomposition based synthesis approach. The algorithm starts with a small set of signals which are certainly needed for the output signal, and uses only the state graphs of the contracted STGs for determining other necessary input signals. Since the state graphs of the contracted STGs are usually very small, it does not suffer from the state explosion problem. Furthermore, its decision procedure computes candidates of the necessary signals in many cases more directly than the greedy algorithm in [17], although some cases need heuristics.

The proposed algorithm, however, has the following restrictions on the class of STGs to be handled. First, the given STG must be 1-safe and output semi-modular, where intuitively, the number of tokens in each place must not exceed one in a 1-safe STG, and output transitions are not disabled by or do not disable any other transitions in an output semi-modular STG. Note that these are required in almost all logic synthesis algorithms. Second, the given STG must have CSC. This is not so restrictive for our purpose, because our compiler from a high-level specification language guarantees it as mentioned before. Finally, the given STG must satisfy the following two properties: (1) the guided simulation with respect to each output signal terminates, and (2) for every two reachable markings of the STG, either one is reachable from the other. These requirements come from the analysis method of the CSC violation traces, and are explained in Section 4. We believe that many specifications to which logic synthesis is applied satisfy these requirements. At least, every benchmark circuit specification shown in Section 5 satisfies them.

The rest of this paper is organized as follows. Section 2 shows the basic theory of our decomposition based synthesis, where several notations are based on [15] and [19]. Section 3 describes the overview of the proposed method, and Section 4 explains in detail how the input sets are determined, which is the main issue of this paper. Several experimental results are shown in Section 5, and Section 6 gives our conclusion.

2. Basic theory

An STG $G = (P, T, F, \mu^0, l, In, Out)$ is a labeled net, where P is a set of *places*, T is a set of *transitions* ($P \cap T = \emptyset$), $F \subseteq (P \times T) \cup (T \times P)$ is the *flow relation*, μ^0 is the *initial marking*, $l : T \to (In \cup Out) \times \{+, -\} \cup \{\lambda\}$ is the labeling function, *In* and *Out* are the input and output signal sets. Let $\mathrm{sig}(G)$ denote $In \cup Out$. A transition t with $l(t) = \lambda$ is called *dummy*. For $w \in \mathrm{sig}(G)$, w-*transition* denotes a transition t with $l(t) = w+$ or $w-$. For any transition t, $\bullet t = \{p \in P \mid (p, t) \in F\}$ and $t\bullet = \{p \in P \mid (t, p) \in F\}$ denote the *source places* and the *destination places* of t, respectively. Transitions t and t' such that $\bullet t \cap \bullet t' \neq \emptyset$ are said to be in *conflict*. Note that when STGs G, G_1, etc. are considered, their corresponding components P, T, etc., P_1, T_1, etc. are implicitly considered.

A *marking* μ of G is any subset of P. A transition t is *enabled* in a marking μ if $\bullet t \subseteq \mu$ (all its source places have tokens in μ); otherwise, it is *disabled*. If a transition t is enabled in μ, it can *fire*, and a new marking $\mu' = (\mu - \bullet t) \cup t\bullet$ is obtained, denoted by $\mu \xrightarrow{t} \mu'$. For a sequence $v = t_1 t_2 \cdots$ of transitions, $\mu \xrightarrow{v} \mu'$ is defined similarly (μ is equal to μ' for an empty v). v is called a *trace*, if there exists μ' such that $\mu^0 \xrightarrow{v} \mu'$. Let $\mathsf{trace}(G)$ denote the set of all traces of G. A trace may contain multiple occurrences of the same transition. In this paper, it is assumed that those occurrences of the same transition are distinguished by some appropriate way, such as, by attaching firing counts.

Each marking has a state vector, which represents the values of signals in $In \cup Out$. Different markings may have the same state vector. In this paper, a *state* implies a state vector or a set of markings with the same state vector. It is sometimes convenient to annotate to a state the information whether the outputs are excited to rise or fall. For this purpose, R or F is used in addition to 0 or 1 in state vectors. R represents the binary value of 0, but it implies that the output is excited to rise. F indicates the signal is 1, but it is excited to fall. When these two notations with and without F/R should be distinguished, we call the former *decorated states*, and the latter *nondecorated states*. For example, suppose that two markings μ and μ' have decorated state (1010) and (101R). They have the common nondecorated state (1010), but the behavior of the output is different in those markings. This situation is called a *CSC violation*, and these two markings are a *CSC violation pair*. If an STG has a CSC violation pair, we say that the STG does not have CSC. Otherwise, it has CSC. If an STG does not have CSC, a circuit cannot be synthesized from the STG.

The property called output semi-modularity is also necessary to synthesize a circuit from an STG. Formally, this property is violated, if and only if there are two chains v_1 and v_2 of dummy transitions such that their first transitions are in conflict, and that either of the non-dummy transitions that follow v_1 or v_2 is related to an output signal. The most simple case is that v_1 and v_2 are empty, and so, an output transition disables or is disabled by an input or output transition.

For $x \in Out$, $ES(x+)$ denotes a set of reachable nondecorated states where x can rise, $QS(x+)$ is a set of reachable nondecorated states where x is stable high. $ES(x-)$ and $QS(x-)$ are defined similarly. The other states are unreachable, and this set is denoted by UR. From the definition of CSC, if and only if an STG has CSC, its $ES(x+)$, $QS(x+)$, $ES(x-)$, and $QS(x-)$ sets are disjoint for each $x \in Out$.

In this paper, the implementation technologies considered are *atomic gates* and *generalized-C (gC) elements*. A circuit for an atomic gate implementation for each $x \in Out$ is defined by a cover $C(x)$, which is a set of states where the logic function of the circuit produces 1. The cover is *correct* with respect to G, if it satisfies

$$C(x) - UR = ES(x+) \cup QS(x+).$$

The gC implementation needs two covers $C(x+)$ and $C(x-)$, and they are *correct* with respect to G, if they satisfy

$$ES(x+) \subseteq C(x+) - UR \subseteq ES(x+) \cup QS(x+),$$
$$ES(x-) \subseteq C(x-) - UR \subseteq ES(x-) \cup QS(x-).$$

If the covers are correct with respect to G, then the corresponding circuit is also *correct* with respect to G.

For a nondecorated state s and a set D of signals, the *D-closure* of s, denoted by $\mathcal{C}_D(s)$, is a set of all nondecorated states, including s, such that their state vectors are the same if the signals in D are projected out. The *core* of a D-closure is the common state vector obtained by projecting out the signals in D. For example, for $s = (abcd) = (1101)$ and $D = \{a, b\}$, $\mathcal{C}_D(s) = \{0001, 0101, 1001, 1101\}$ and its core is $(cd) = (01)$. The mappings from D-closure $\mathcal{C}_D(s)$ to its core s' and its inverse are defined by $\mathsf{proj}_D(\mathcal{C}_D(s))$ and $\mathsf{proj}_D^{-1}(s')$. Note that both are the one-to-one mappings. The D-closure and these mappings are extended to sets as follows: $\mathcal{C}_D(S) = \bigcup_{s \in S} \mathcal{C}_D(s)$, $\mathsf{proj}_D(\mathcal{C}_D(S)) = \{\mathsf{proj}_D(\mathcal{C}_D(s)) \mid s \in S\}$, and $\mathsf{proj}_D^{-1}(S') = \bigcup_{s' \in S'} \mathsf{proj}_D^{-1}(s')$.

For an STG G and $x \in Out$, a set D of signals is an *irrelevant input set* for x, if

1. $D \subseteq In \cup Out - \{x\}$,
2. $\mathcal{C}_D(ES(x+)) - UR = ES(x+)$, and
3. $\mathcal{C}_D(ES(x-)) - UR = ES(x-)$.

From this definition, the following lemma holds.

Lemma 1 For an STG G with CSC and an irrelevant input set D for its output x, $\mathcal{C}_D(ES(x+))$, $\mathcal{C}_D(QS(x+))$, $\mathcal{C}_D(ES(x-))$, and $\mathcal{C}_D(QS(x-))$ are disjoint.

The proof is shown in the appendix. From this lemma and $\mathcal{C}_D(S) \supseteq S$ for any set S, $QS(x+)$ and $QS(x-)$ satisfy

1. $\mathcal{C}_D(QS(x+)) - UR = QS(x+)$, and
2. $\mathcal{C}_D(QS(x-)) - UR = QS(x-)$,

when G has CSC and D is an irrelevant input set.

For an STG G and a set D of signals, let G_D denote an STG obtained from G by making transitions related to signals in D dummy. Then, the following lemma holds.

Lemma 2 Suppose that an STG G has CSC and is output semi-modular. For any $x \in Out$ and any irrelevant input set D for x, a speed-independent circuit for x synthesized from G_D is correct with respect to G.

Intuitively, this can be explained as follows. From the above properties for $\mathcal{C}_D(ES(x+))$, $\mathcal{C}_D(ES(x-))$, $\mathcal{C}_D(QS(x+))$, and $\mathcal{C}_D(QS(x-))$, even if the values of signals in D are changed in a state, the resulting state falls in the same state set (i.e., $ES(x+)$, $ES(x-)$, $QS(x+)$, or $QS(x-)$) as the original state, if it is reachable. Hence, the behavior of an STG is not affected by projecting out the signals in D. A more formal proof is shown in the appendix.

On the other hand, for a non-irrelevant input set D, G_D no longer has CSC as shown below.

Lemma 3 Suppose that an STG G has CSC, and for $x \in Out$, a set D of signals with $D \subseteq In \cup Out - \{x\}$ is not an irrelevant input set for x. Then, G_D does not have CSC with respect to x.

(Proof) Since D is not an irrelevant input set, there exist either states $s \in ES(x+)$ and $s' \in \mathcal{C}_D(s) - UR$ such that $s' \notin ES(x+)$, or states $s \in ES(x-)$ and $s' \in \mathcal{C}_D(s) - UR$ such that $s' \notin ES(x-)$. In the former case, s' must be in $QS(x+) \cup ES(x-) \cup QS(x-)$. But, the value of x in s is 0, and so is the value of x in s' from $x \notin D$. Thus, $s' \in QS(x-)$ holds. s and s' are mapped in the same state in G_D. Hence, $s \in ES(x+)$ and $s' \in QS(x-)$ imply that G_D has a CSC violation with respect to x. The similar discussion holds for the latter case. (Q.E.D.)

For STGs G_1 and G_2, a *simulation* from G_1 to G_2 is a relation S between markings of G_1 and G_2 satisfying

- $(\mu_1^0, \mu_2^0) \in S$, and
- for all $(\mu_1, \mu_2) \in S$ and all $\mu_1 \xrightarrow{v} \mu_1'$ with $v = t_1 t_2 \cdots t_n (n \geq 0)$, there exists some firing sequence $v' = t_1' t_2' \cdots t_m' (m \geq 0)$ and μ_2' such that $\mu_2 \xrightarrow{v'} \mu_2'$, $l_1(v) = l_2(v')$ and $(\mu_1', \mu_2') \in S$, where for $u = u_1 u_2 \cdots u_k (k \geq 0)$, $l(u)$ is obtained from $l(u_1) l(u_2) \cdots l(u_k)$ by deleting λ.

If B is a simulation from G_1 to G_2, and B^{-1} is a simulation from G_2 to G_1, then B is a *bisimulation* between G_1 and G_2. Let $G_1 \rightsquigarrow G_2$ and $G_1 \approx G_2$ denote that there exist a simulation from G_1 to G_2 and a bisimulation between G_1 and G_2, respectively.

For an STG G, $x \in Out$, and $V \subset \mathsf{sig}(G)$ such that $x \in V$, $\mathsf{abs}(G, V, x)$ is any STG with the input signal set $V - \{x\}$ and the output signal set $\{x\}$ such that $\mathsf{abs}(G, V, x) \approx G_D$ with $D = \mathsf{sig}(G) - V$. $\mathsf{abs}(G, V, x)$ can be obtained by the net contraction algorithm, and it can usually be constructed such that its state space is much smaller than that of G. The details can be found, for instance, in [15].

Theorem 1 Suppose that an STG G has CSC and is output semi-modular. For $x \in Out$ and some $V \subset \mathsf{sig}(G)$ with $x \in V$, if $\mathsf{abs}(G, V, x)$ has CSC, then a speed-independent circuit for x synthesized from $\mathsf{abs}(G, V, x)$ is correct with respect to G.

(Proof) If $D = \mathsf{sig}(G) - V$ is not an irrelevant input set for x, then from Lemma 3, G_D does not have CSC with respect to x. This implies that $\mathsf{abs}(G, V, x)$ does not

```
decomposition_based_synthesis(G) {
  forall x ∈ Out {
    G_abs = obtain_synthesizable_abs(G,x)
    if (G_abs == ''impossible'') then abort
    C_x = logic_synthesis(G_abs)
  }
}
```

Figure 1. Top-level algorithm for synthesis.

have CSC either, because $\mathsf{abs}(G,V,x) \approx G_D$. Hence, D is an irrelevant input set for x. From Lemma 2, a correct speed-independent circuit for x is synthesized from G_D. Again, from the bisimilarity between G_D and $\mathsf{abs}(G,V,x)$, $\mathsf{abs}(G,V,x)$ produces the same circuit as the one obtained from G_D. (Q.E.D.)

From this theorem, if an input set V such that $\mathsf{abs}(G, V, x)$ has CSC is determined, a correct speed-independent circuit for x can be synthesized efficiently. The main contribution of this work is to develop its decision procedure without constructing the state graph of G.

3. Decomposition based synthesis overview

The top level algorithm for the proposed decomposition based synthesis is shown in Figure 1. It tries to compute a synthesizable abstraction G_{abs} for each output signal x of G. This is actually $\mathsf{abs}(G,V,x)$ that has CSC for some V. If it is impossible, then it is proven in the theorem shown later that G does not have CSC, and so the algorithm aborts. Otherwise, an ordinary speed-independent logic synthesis tool such as Petrify or ATACS is applied to G_{abs} to synthesize a circuit for x.

The algorithm for obtaining synthesizable abstraction is shown in Figure 2. It first constructs the initial input set for x by taking the signals that make x enabled, called *trigger signals* for x [1]. This is because trigger signals belong to no irrelevant input sets as shown in the proof of Lemma 2.

For this initial input set V, the algorithm next computes $\mathsf{abs}(G,V,x)$ [2], and check if it has CSC. If it does, the algorithm returns it. Otherwise, some set of traces of $\mathsf{abs}(G,V,x)$ that cause CSC violations, $CSCV$, is extracted [3] by generating and checking the state graph of $\mathsf{abs}(G,V,x)$. Note that this state graph is usually much smaller than that of the original G. The algorithm then analyzes each $g \in CSCV$ and tries to find candidate inputs

1 This is actually implemented in a conservative way such that it takes the signals related to the first non-dummy transitions reached from $x+$ or $x-$ transitions by the upward net traversal of G.

2 A simplified version of the algorithm shown in [15] is used to compute $\mathsf{abs}(G,V,x)$.

3 In our current implementation, one shortest trace is selected for each CSC violation pair, because using all CSC violation traces is very expensive.

```
obtain_synthesizable_abs(G,x) {
  V = initial_input_set(G,x)
  loop {
    G_abs = obtain_abs(G,V,x)
    if (G_abs has CSC) then return G_abs
    CSCV =
        obtain_CSC_violation_trace_set(G_abs)
    forall g ∈ CSCV {
      candidate =
        analyze_CSCV_trace(g,G,V,x,μ0,null)
      if (candidate == ''impossible'') then
        return ''impossible''
      add_constraints_matrix(candidate)
    }
    newV = solve_covering_problem()
    V = V ∪ newV
  }
}
```

Figure 2. Algorithm to obtain synthesizable abstraction.

to be added in order to resolve the CSC violation. The algorithm may decide that it is impossible to resolve the CSC violation by adding any signals in G. In this case, the algorithm returns "impossible". Otherwise, *candidate* contains a set of requirements such that each requirement is a set of signals, and it is satisfied if at least one of them is added to V. In order to resolve the CSC violation, every requirement must be satisfied. Those requirements are added in the constraint matrix to set up a covering problem. This process is repeated for every CSC violation trace in CSCV. Finally, the covering problem is solved for those requirements, and the optimal set of signals are added to V. This V is used to compute a new $\mathsf{abs}(G,V,x)$, and the algorithm repeats the above process.

4. Analyzing CSC violation trace

This section shows the detailed algorithm for **analyze_CSCV_trace** that analyzes the CSC violation traces and determines a set of requirements for an appropriate input set for x. As shown in Figure 2, **analyze_CSCV_trace** receives g as a parameter. This g is a trace of $\mathsf{abs}(G,V,x)$ that causes a CSC violation. In order to find an appropriate input set for x, it is necessary to obtain a concrete trace of G that corresponds to g. This can be done without generating the state graph of G by a technique similar to the one used for the partial order reduction, which we call *guided simulation*. This section starts with the guided simulation algorithm, and then shows how to determine the candidates for the input. In the following, an *interface signal* means the signals used in $\mathsf{abs}(G,V,x)$, i.e., the signals in V, and a *noninterface signal* means the remaining sig-

```
analyze_CSCV_trace(g, G, V, x, μ, f) {
  if (g is empty) {
    candidate = find_inputs(f, G, V, x)
    return candidate
  }
  g1 = pick the first transition of g
  N = find_firing_trans(μ, g1)
  if (N is empty) return ''backtrack''
  forall t ∈ N {
    μ' = fire(μ, t)
    g' = g
    if (t == g1)
      remove first transition of g'
    result =
      analyze_CSCV_trace(g', G, V, x, μ', f·t)
    if (result ≠ ''backtrack'')
      exit forall
  }
  return result
}
```

Figure 3. Algorithm for the guided simulation.

```
find_firing_trans(μ, t) {
  if (t ∈ enabled(μ))
    result = {t} ∪ (enabled(μ) ∩ NonIF)
  else result =necessary(μ, t)
  if (∃u ∈ result ∩ NonIF s.t. conflict(u) = {u})
    return {u}
  if (result − conflict(t) ≠ ∅)
    return (result − conflict(t))
  return result
}

necessary(μ, t) {
  if (t is already visited) return ∅
  if (t ∈ enabled(μ)) return {t}
  result = the set of all transitions
  forall (p ∈ •t − μ) {
    π = ∅
    forall (t' ∈ •p ∩ NonIF)
      π = π ∪ necessary(μ, t')
    if (π is smaller than result)
      result = π
  }
  return result
}
```

Figure 4. Algorithms for finding transitions to be fired and for constructing necessary sets.

nals of G, i.e., the signals in $D = \mathsf{sig}(G) - V$. The corresponding transitions are called similarly.

4.1. Guided simulation

For a given abstracted trace g, the guided simulation obtains a trace f of G such that a trace obtained by projecting out the noninterface signals from f is equal to g. More formally, $\mathsf{del}(D, f) = g$ with $D = \mathsf{sig}(G) - V$, where for $h = t_1 t_2 t_3 \cdots$ and a set E of signals, if $l(t_1) \in E \times \{+, -\}$, then $\mathsf{del}(E, h) = \eta$, otherwise, $\mathsf{del}(E, h) = t_1 \eta$, with $\eta = \mathsf{del}(E, t_2 t_3 \cdots)$.

Although an interface transition and a noninterface transition that are concurrent can fire in any order, our algorithm fires all possible noninterface transitions before firing an interface transition because this greatly simplifies the analysis algorithm shown later. We call a trace satisfying this property a *regular* trace. This simplification can, however, cause situations where the guided simulation does not terminate. This happens, for example, when there exists a loop in which noninterface transitions can fire forever. Practically, such situations can be detected by counting the continuous firings of noninterface transitions and checking if it exceeds some upper bound.

The algorithm for the guided simulation is shown in Figure 3. It forms the body of the recursive procedure **analyze_CSCV_trace**. If g is nonempty, the algorithm picks the first transition of g, denoted by g_1, and computes by **find_firing_trans**(μ, g_1) a set of transitions that should be fired for g_1. Figure 4 shows the algorithm for **find_firing_trans**. It first computes a set of potentially necessary transitions for g_1, which is $\{g_1\} \cup (enabled(\mu) \cap NonIF)$ if g_1 is enabled, where $NonIF$ is a set of all noninterface transitions, and **necessary**(μ, g_1) otherwise. The former is for satisfying regularity, and the latter is a minimal set of enabled noninterface transitions such that g_1 can never be enabled if none of those transitions is fired. For example, consider the nets shown in Figure 5, where the transitions except for g_1 are noninterface. In the case of Figure 5(a), the necessary set of g_1 is $\{t_1\}$ or $\{t_2\}$. $\{t_1\}$ can be a necessary set of g_1, because g_1 cannot be enabled if t_1 is not fired. Similarly, $\{t_2\}$ can be another necessary set of g_1. On the other hand, in the case of Figure 5(b), even if t_1 is not fired, g_1 may be enabled by a token produced by firing t_2. If neither t_1 nor t_2 is fired, g_1 cannot be enabled. Thus, the necessary set of g_1 for this case is $\{t_1, t_2\}$. **find_firing_trans** then chooses a set of transitions that should actually be fired. The first two conditions are for firing enabled noninterface transitions earlier than g_1 in order to satisfy regularity. If $result$ contains an enabled noninterface transition that conflicts with no other transitions (i.e., $conflict(u) = \{u\}$, where $conflict(u) = \{u' \mid {\bullet u} \cap {\bullet u'} \neq \emptyset\}$), then it can be fired alone. Otherwise, all conflicting transitions except for $conflict(g_1)$ are returned and used for backtracking. Finally, if there exist no transitions that are concurrent with g_1, g_1 and its conflicting enabled noninterface transitions are returned.

If **find_firing_trans** returns more than one transi-

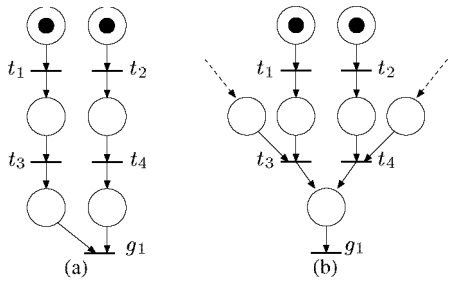


Figure 5. Examples for the necessary set construction.

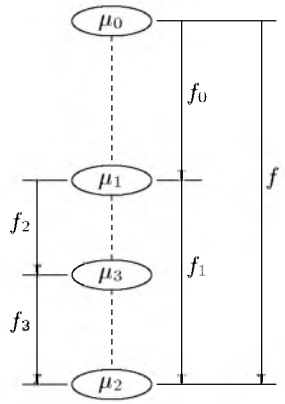


Figure 6. Labeling of a concrete trace.

tion, then those transitions are fired one by one in **analyze_CSCV_trace**. This is because it is impossible to find the exact transition that should be fired in the marking, and so, the algorithm relies on backtracking. If an inappropriate transition is fired, then it becomes impossible to fire the next transition of g in some marking, and an empty set is returned by **find_firing_trans**. In this case, backtracking occurs by returning "backtrack".

Although it is guaranteed by this backtracking mechanism that the concrete trace that corresponds to g is certainly obtained, G with many conflicts sometimes causes a lot of backtrackings. Our tool supports two options that we consider to be practical solutions for this problem. One is to keep every conflicting transition in $\mathsf{abs}(G, V, x)$, and the other is to keep some of those conflicting transitions, which are selected by the users through specific comments in the STG file. When translating specifications written in a high-level language, our compiler automatically suggests the selection of conflicting transitions that should be kept in $\mathsf{abs}(G, V, x)$ by using the latter option.

The fired transitions are appended to f in each recursive call of **analyze_CSCV_trace**. When g becomes empty, f holds the corresponding concrete trace, which is passed to **find_inputs**.

4.2. Determining the input set

Each CSC violation trace g of $\mathsf{abs}(G, V, x)$ constructed by **obtain_CSC_violation_trace_set** is assumed to satisfy that $g = \langle g_0, g_1 \rangle$, $\mu_0' \stackrel{g_0}{\rightarrow} \mu_1'$ (μ_0' is the initial marking of $\mathsf{abs}(G, V, x)$), $\mu_1' \stackrel{g_1}{\rightarrow} \mu_2'$, μ_1' and μ_2' are a CSC violation pair, and there are no markings between μ_1' and μ_2' that have the same nondecorated state as that of μ_1'. The corresponding concrete trace f generated by the guided simulation is also of the form $\langle f_0, f_1 \rangle$ such that f_0 and f_1 end by interface transitions. Let μ_1 and μ_2 denote the markings of G obtained by f_0 and f_1, respectively (see Figure 6). Note that this assumption simplifies the input set decision procedure, but in order to guarantee that every CSC violation pair is reached from the initial marking by a simple path, the STG needs to satisfy the property that for its every two reachable markings, either one is reachable from the other.

For two interface transitions a and b in f, if a fires before b, it is denoted by $(a, b) \in R^f_{order}$. For two (interface or noninterface) transitions t_1 and t_2 in f, if t_1 causes t_2, that is, t_2 fires by consuming the token produced by the firing of t_1, it is denoted by $(t_1, t_2) \in R^f_{cause}$. If t_1 and t_2 are related by the transitive closure of the union of R^f_{order} and R^f_{cause}, i.e., $(t_1, t_2) \in (R^f_{order} \cup R^f_{cause})^*$, then we say that t_1 is an *ancestor* of t_2 in f, and denoted by $[t_1 \rightsquigarrow_f t_2]$. Since the specific abstracted trace g is focused on, this ancestor relation represents an actual causality relation with respect to g. The reason why the ordering relation of interface signals is also included is that if the firing order of concurrent interface transitions in g changes, then it is considered to be a different CSC violation trace with a different CSC violation state pair, and such a different trace is handled separately in the forall loop of **obtain_synthesizable_abs**. This ancestor relation plays a key role in the proposed algorithm. Thus, in the first step of **find_input**, this ancestor relation is set up, which is actually done by constructing a data structure similar to an occurrence net [20].

In order to resolve the CSC violation between μ_1 and μ_2, it is necessary to find a noninterface signal w such that f_1 contains odd number of w-transitions. If w-transition certainly fires in f_1 in odd times, then the CSC violation is resolved. However, the existence of concurrent transitions makes this decision difficult. Thus, we need to define the following notions (see Figure 7).

$\mathsf{final}(h)$ denotes the last transition in h, and $\mathsf{chain}(w, h)$ is a sequence $(t_0 t_1 t_2 \cdots t_{n-1})$ of all w-transitions (i.e., $l(t_i) = w+$ or $w-$ for each i) firing in h in this order. For a trace $h = \langle h_1, h_2 \rangle$ such that at least h_2 ends with an interface transition, and a noninterface signal w, w is *confined* by h_2 in h, if

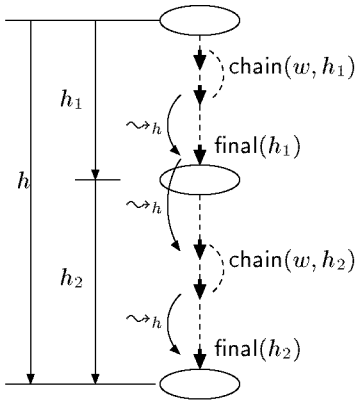


Figure 7. Confinement relation.

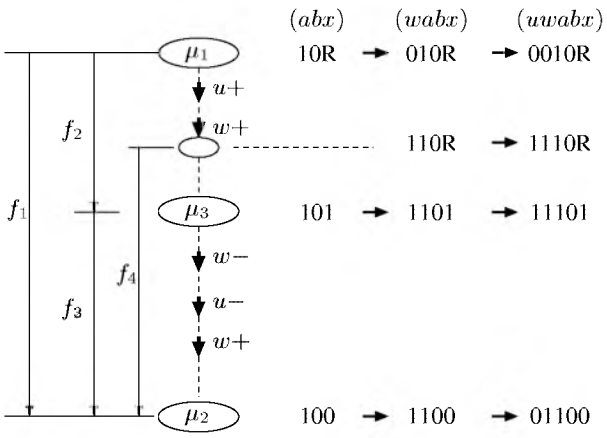


Figure 8. Resolving CSC violation by adding essential signals.

- when w-transition fires in h_1, for the last transition t_{e_1} of chain(w, h_1), $[t_{e_1} \rightsquigarrow_h \text{final}(h_1)]$, and
- when w-transition fires in h_2, for the first transition t_{s_2} and the last transition t_{e_2} of chain(w, h_2), $[\text{final}(h_1) \rightsquigarrow_h t_{s_2}]$ and $[t_{e_2} \rightsquigarrow_h \text{final}(h_2)]$.

If w is confined by h_2 in h, it is guaranteed that all w-transitions that can fire between final(h_1) and final(h_2) are just those in chain(w, h_2). From the regularity and the assumption that h_2 ends with an interface transition, any non-interface transition that is concurrent with final(h_2) fires before final(h_2). Thus, final(h_2) is an ancestor of the next w-transition that fires after h_2. From this reason, it's not necessary to consider the first w-transition after h_2. In the case of $f = \langle f_0, f_1 \rangle$, f_0 also ends with an interface signal. Thus, the condition $[\text{final}(h_1) \rightsquigarrow_h t_{s_2}]$ is not necessary either. The other cases shown later, however, need this condition.

If w is confined by h_2 in h, and chain(w, h_2) contains an odd number of transitions, then w is *odd-confined* by h_2 in h. *even-confined* is defined similarly.

Consider the first interface transition in f_1, and divide f_1 into f_2 and f_3 with it, i.e., $f_1 = \langle f_2, f_3 \rangle$ and f_2 ends with this first interface transition. Figure 6 shows the relation among $f_0 \cdots f_3$. The following lemma holds.

Lemma 4 The CSC violation with respect to f is resolved by adding a noninterface signal w to V, if w is odd-confined by f_1 in f, and f_2 contains no w-transitions.

(Proof) Since w is odd-confined by f_1 in f, it is guaranteed by the ancestor relation that the state vector of μ_1, projected to $V \cup \{w\}$, is different from that of μ_2. Furthermore, from both the assumption with respect to g that there are no markings between μ_1' and μ_2' that have the same binary state vector as μ_1', and the assumption that f_2 contains no w-transitions, no new CSC violation pair is introduced. (Q.E.D.)

In cases that f_2 contains w-transitions, every marking obtained by a w-transition in f_2 has the same state vector, if it is projected to V, because μ_3 is obtained by the first interface transition in f_1. Thus, those markings in odd positions cause a CSC violation with μ_2, because their values of w are the same as that of μ_2. One example is shown in Figure 8. The first $w+$ in f_2 leads to a marking with state vector $(110R)$, and this state has a CSC violation with μ_2. This CSC violation cannot be resolved only using the signal w.

Such a CSC violation, however, can be resolved by adding another noninterface signal u in addition to w such that u is odd-confined by f_4 in f and the first u-transition in f_4 fires in f_3, where f_4 is the suffix of f_1 after the first w-transition. The final column of Figure 8 shows how the CSC violation is resolved in this case. This additional signal u is called *essential* for w in f. Hence, in this case, a signal w together with its essential signal u resolves the CSC violation. The precise definition of essential signals is shown as follows.

Suppose that w is odd-confined by f_1 in f, and chain$(w, f_1) = t_1 t_2 \cdots t_n$, where t_k is the last w-transition that fires in f_2. For every odd integer $i \leq k$, a noninterface signal $u_i (\neq w)$ is essential for w in f with respect to i, if (1) u_i is odd-confined by h_i in f, where h_i is the suffix of f_1 after t_i, and (2) for the first u_i-transition t_{u_i} in h_i,

- if either t_{i+1} does not exist or $[\text{final}(f_2) \rightsquigarrow_f t_{i+1}]$, then $[\text{final}(f_2) \rightsquigarrow_f t_{u_i}]$,
- else, $[t_{i+1} \rightsquigarrow_f t_{u_i}]$

holds. A set of essential signals that includes some essential signal u_i for every odd integer $i \leq k$ is called a *sufficient essential signal set* for w.

Figure 9 shows a sufficient essential signal set $\{u_1, u_3\}$ for w. Each essential signal distinguishes the (common) state vector of all the markings between the one obtained by

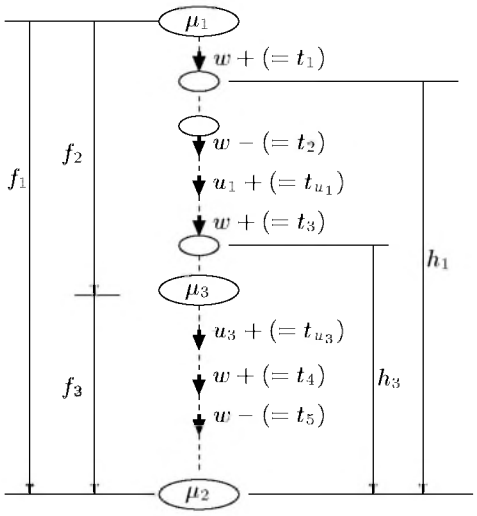
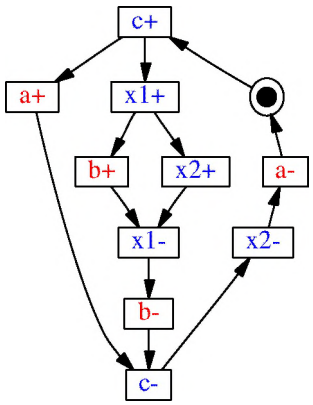


Figure 9. An example where more than one essential signal is necessary.



INPUTS: a,b
OUTPUTS: c,x1,x2

Figure 10. An STG where our sufficient condition does not work.

t_i and the one where t_{i+1} fires from that of μ_2. For this reason, the first u-transition t_{u_i} must fire certainly after t_{i+1}. Furthermore, final(f_2) plays the role of t_{i+1} in the last section of f_2. From these discussions, the following theorem, which is a generalized version of Lemma 4, holds.

Theorem 2 The CSC violation with respect to f is resolved by adding a noninterface signal w that is odd-confined by f_1 in f and its sufficient essential signal set to V.

Note that this is a sufficient condition. Thus, even if the signals satisfying the above condition are not found, CSC violations may be resolved. For example, consider the STG G shown in Figure 10 and its output c. The trigger signals for c is a and b. abs$(G, \{a, b, c\}, c)$ has a CSC violation trace $g = c+, a+, b+, b-$ with $g_0 = c+, a+$ and $g_1 = b+, b-$. The guided simulation generates $f_0 = c+, x1+, x2+, a+$ and $f_1 = b+, x1-, b-$. $x1$-transitions appear odd-times in f_1. $x1$ is, however, not confined by f_1 in f, because $x1+$ and $a+$ are concurrent, and so, $[x1+ \not\rightsquigarrow_f a+]$. There exists no other noninterface signal that is confined by f_1. Hence, no signals do not satisfy the above condition. However, G itself has CSC. Thus, although $x1$ and $x2$ do not satisfy our sufficient condition, $x1$ together with $x2$ can resolve the CSC violation.

On the other hand, it is possible to decide that a given CSC violation can never be resolved by adding any signals. One sufficient condition is as follows.

Theorem 3 If there exists a (noninterface or interface) transition t in f_2 such that $[t \rightsquigarrow_f \text{final}(f_2)]$, and for any noninterface signal u, u is even-confined by the suffix of f_1 from t in f, then G has no CSC (see Figure 11).

(Proof) The state vector of the marking where t fires is the

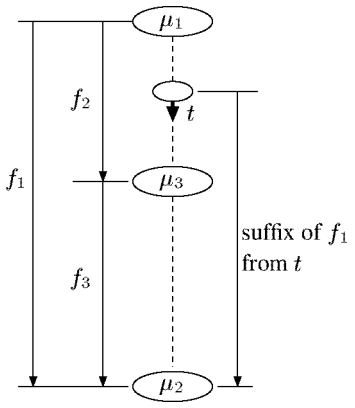


Figure 11. Unresolvable CSC violation.

same as that of μ_2. Furthermore, from $[t \rightsquigarrow_f \text{final}(f_2)]$ and the regularity, they certainly cause a CSC violation. Since for any noninterface signal u, u is even-confined, any noninterface signal cannot distinguish those state vectors. Hence, this CSC violation cannot be resolved even by adding all noninterface signals. This means that G has no CSC. (Q.E.D.)

If the above condition is satisfied, **find_input** returns "impossible". Otherwise, it tries to find noninterface signals that can resolve the given CSC violation. If it succeeds, there are usually many choices. For example, suppose that signals a, b, c, and d can be w, and c has a sufficient essential signal set $\{e\}$, and d has a sufficient essential signal set $\{f\}$ or $\{g\}$. The whole condition can be expressed by

$$a \vee b \vee (c \wedge e) \vee (d \wedge (f \vee g)).$$

Table 1. Experimental results (1).

Circuit	(#I,#O)	Petrify			Proposed method			
		CPU (s)	Mem(MB)	area	CPU(s) (Petrify+other)	Max(MB)	area	ave. #L
cb	(10,10)	9.6	4.6	82	3.5 = (3.3+0.2)	3.4	82	1.2
cachem	(11,16)	219.5	7.7	122	36.8 = (36.1+0.7)	4.0	123	1.5
lf6	(21,41)	† (≥59272.4)	(≥742)	–	98.9 = (94.9+4.1)	4.7	200	1.2

(† : BDD manager overflow: ≥ 30000000 nodes)

Table 2. Experimental results (2).

Circuit	(#I,#O)	Petrify			Proposed method		
		CPU (s)	Mem(MB)	area	CPU(s) (Petrify+other)	Max(MB)	area
FIR5_2mul_csc	(7,19)	78.8	7.8	151	32.4 = (31.2+1.2)	4.3	150
IIR2_2mul_d_csc	(7,19)	240.2	12.3	150	179.6 = (176.9+2.7)	5.8	151
LMS4_pr12_csc	(9,18)	354.6	18.6	177	28.3 = (26.6+1.7)	4.5	177

To set up the covering problem, this is transformed into product-of-sum form such as

$$(a \vee b \vee c \vee d)(a \vee b \vee c \vee f \vee g)(a \vee b \vee e \vee d)(a \vee b \vee e \vee f \vee g).$$

Each clause is a requirement (see Section 3), and **find_input** returns a set of those requirements.

As long as the condition of Theorem 3 does not hold, even if no noninterface signals satisfy the sufficient condition for resolving the CSC violation, it is worth adding some signals as shown in the example of Figure 11. **find_input** uses some heuristics to choose those signals, for instance, choosing a noninterface signal of which transitions just appear odd-times in f_1.

5. Experimental results

The proposed method has been implemented using the C language. This section evaluates the potential performance of the proposed method and the quality of the synthesized circuits. The experiments here have been done on a 2.8 GHz Pentium 4 workstation with 4 gigabytes of memory. As for the selection of conflicting transitions in order to avoid backtracking, the compiler suggestion is used for the first set of examples, and the option to keep every conflicting transition is used for the second and third sets. Note that the performance of Petrify shown in these experiments is just used for suggesting the complexity of each example, not for comparing the superiority of both methods. On the other hand, the estimated area sizes shown by Petrify are a good criterion to evaluate the overhead of our method.

The first set of experimental results is shown in Table 1. In this experiment, the STGs generated from a high-level specification language by the method shown in [12] are used. Those are for the sub-circuits of the instruction cache system for TITAC2 asynchronous microprocessor [21]. These STGs originally contain many dummy transitions generated by the compiler from the high-level specification language, and those dummy transitions degrade the performance of synthesis process by Petrify or contraction process in our method. Thus, the reduced STGs obtained by removing those dummy transitions from the original STGs are used. The second column of the table shows the number of input signals and output signals of the circuits. The third main column shows the CPU times, the memory usage, and the estimated area of the synthesized circuits by Petrify with gC implementation option. Petrify cannot synthesize "lf6" due to BDD manager overflow. The fourth main column shows the results of our method. For fair comparison, Petrify is used for the logic synthesis of each ab-

Table 3. Experimental results (3).

Circuit	(#I,#O)	Petrify		Proposed method	
		CPU (s)	area	CPU(s)	area
alloc-outbound	(4,5)	0.05	21	0.12	21
atod	(3,4)	0.06	23	0.18	23
chu150	(3,3)	0.06	21	0.12	21
chu172	(3,3)	0.03	7	0.06	7
converta	(2,3)	0.06	28	0.19	28
dff	(2,2)	0.05	14	0.14	14
master-read	(6,8)	2.23	53	1.72	53
mp-forward-pkt	(12,3)	0.12	29	0.08	29
nak-pa	(4,6)	0.14	31	0.23	31
nowick	(3,3)	0.08	25	0.10	25
pe-rcv-ifc	(4,7)	1.68	64	1.61	65
pe-send-ifc	(5,5)	1.38	64	1.51	64
ram-read-sbuf	(5,6)	0.23	32	0.28	32
rcv-setup	(3,2)	0.03	14	0.12	14
sbuf-ram-write	(5,7)	0.27	30	0.40	30
sbuf-read-ctl	(3,5)	0.08	19	0.13	19
sbuf-send-pkt2	(4,5)	0.15	25	0.33	25
sendr-done	(2,2)	0.02	7	0.03	7
trimos-send	(3,6)	0.38	48	1.38	48
vbe10b	(4,7)	0.30	47	0.62	47
vbe4a	(3,3)	0.06	9	0.05	9
vbe6a	(4,4)	0.22	42	0.66	42
vmebus-arb	(3,2)	0.23	14	0.46	14
wrdata	(1,4)	0.03	22	0.19	22
wrdatab	(4,6)	0.43	53	0.86	53

stracted STG, e.g., Petrify is run 41 times to synthesize the sub-circuit for each output in the case of "lf6", although any logic synthesis tool can be used. Our CPU times show both the run times of this final logic synthesis process by Petrify and the remaining run times for contraction, state space generation of abstracted STGs, and analysis of CSC violation traces. The final sub-column shows the average loop counts in **obtain_synthesizable_abs**.

These results show that our method efficiently handle a large specification that the traditional method cannot handle. As for the area overhead, it seems small, but since the large example cannot be synthesized by Petrify, more comparisons are necessary.

For this purpose, some examples from [22] and the standard benchmark examples are used. Table 2 and Table 3 show these results. From these results, the area overhead is just 0.2%. Thus, even though our method uses restricted information for synthesizing sub-circuits, the quality at least with respect to the area size is not badly affected.

6. Conclusion

This paper presents a decomposition method for efficient synthesis of very large speed-independent circuits. While this method does have some overhead for small circuits, it allows for the synthesis of large circuits that could not be synthesized using flat synthesis methods. The experimental results show that the area overhead appears to be very small.

Although the theory and algorithms presented in this paper are for untimed circuits, the proposed idea can be extended to timed circuit synthesis by minor modification as follows. Let G_{timed} be a timed STG, where transitions have the earliest and latest firing times, and $G_{untimed}$ be its untimed version (i.e., a STG obtained from G_{timed} by removing all earliest and latest firing times). If $G_{untimed}$ satisfies the requirements for our method, it is safe to apply our decision procedure to $G_{untimed}$ in order to obtain the necessary signals of G_{timed}. Since the decision procedure does not use timing information, the input signal sets may not be optimal, i.e., some redundant signals may be included, but necessary signals are never missed. Thus, if the resultant input sets are used by the timed contraction of G_{timed} followed by the timed logic synthesis procedure, an optimized circuit using the timing information can be synthesized for G_{timed}. The timed circuit synthesis using this idea from high-level specification languages together with its compiler is future work as well as the comparison with other approaches like syntax directed translation.

Acknowledgment

We'd like to thank H. Saito for giving us his benchmark circuits. We also thank the referees for their helpful comments.

References

[1] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev. Petrify: a tool for manipulating concurrent specifications and synthesis of asynchronous controllers. *IEICE Transactions on Information and Systems*, E80-D(3):315–325, March 1997.

[2] P. A. Beerel, C. J. Myers, and T. H.-Y. Meng. Covering conditions and algorithms for the synthesis of speed-independent circuits. *IEEE Transactions on Computer-Aided Design*, March 1998.

[3] R. M. Fuhrer, S. M. Nowick, M. Theobald, N. K. Jha, B. Lin, and L. Plana. Minimalist: An environment for the synthesis, verification and testability of burst-mode asynchronous machines. Technical Report TR CUCS-020-99, Columbia University, NY, July 1999.

[4] Steven M. Burns and Alain J. Martin. Syntax-directed translation of concurrent programs into self-timed circuits. In J. Allen and F. Leighton, editors, *Advanced Research in VLSI*, pages 35–50. MIT Press, 1988.

[5] Kees van Berkel, Joep Kessels, Marly Roncken, Ronald Saeijs, and Frits Schalij. The VLSI-programming language Tangram and its translation into handshake circuits. In *Proc. European Conference on Design Automation (EDAC)*, pages 384–389, 1991.

[6] Euiseok Kim, Jeong-Gun Lee, and Dong-Ik Lee. Automatic process-oriented control circuit generation for asynchronous high-level synthesis. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 104–113. IEEE Computer Society Press, April 2000.

[7] Joep Kessels and Ad Peeters. The Tangram framework: Asynchronous circuits for low power. In *Proc. of Asia and South Pacific Design Automation Conference*, pages 255–260, February 2001.

[8] Doug Edwards and Andrew Bardsley. Balsa: An asynchronous hardware synthesis language. *The Computer Journal*, 45(1):12–18, 2002.

[9] A. Bystrov and A. Yakovlev. Asynchronous circuit synthesis by direct mapping: Interfacing to environment. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 127–136, April 2002.

[10] Tiberiu Chelcea and Steven M. Nowick. Resynthesis and peephole transformations for the optimization of large-scale asynchronous systems. In *Proc. ACM/IEEE Design Automation Conference*, June 2002.

[11] Chris J. Myers and Teresa H.-Y. Meng. Synthesis of timed asynchronous circuits. *IEEE Transactions on VLSI Systems*, 1(2):106–119, June 1993.

[12] Tomohiro Yoneda and Chris Myers. Synthesizing timed circuits from high level specification languages. *NII Technical Report*, NII-2003-003E, 2003.

[13] A. Matsumoto. High level synthesis of asynchronous circuits (in Japanese). *Master Thesis, Tokyo Insitute of Technology*, 2004.

[14] Tam-Anh Chu. *Synthesis of Self-Timed VLSI Circuits from Graph-Theoretic Specifications*. PhD thesis, MIT Laboratory for Computer Science, June 1987.

[15] Walter Vogler and Ralf Wollowski. Decomposition in asynchronous circuit design. In J. Cortadella, A. Yakovlev, and

G. Rozenberg, editors, *Concurrency and Hardware Design*, volume 2549 of *Lecture Notes in Computer Science*, pages 152–190. Springer-Verlag, 2002.

[16] H. Zheng, E. Mercer, and C. J. Myers. Modular verification of timed circuits using automatic abstraction. *IEEE Transactions on Computer-Aided Design*, 22(9), September 2003.

[17] Ruchir Puri and Jun Gu. A modular partitioning approach for asynchronous circuit synthesis. In *Proc. ACM/IEEE Design Automation Conference*, pages 63–69, June 1994.

[18] J. Beister, G. Eckstein, and R. Wollowski. From STG to extended-burst-mode machines. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 145–158, April 1999.

[19] Chris Myers. *Asynchronous Circuit Design*. John Wiley & Sons, 2001.

[20] Kenneth McMillan. Using unfoldings to avoid the state explosion problem in the verification of asynchronous circuits. In G. v. Bochman and D. K. Probst, editors, *Proc. International Workshop on Computer Aided Verification*, volume 663 of *Lecture Notes in Computer Science*, pages 164–177. Springer-Verlag, 1992.

[21] Akihiro Takamura, Masashi Kuwako, Masashi Imai, Taro Fujii, Motokazu Ozawa, Izumi Fukasaku, Yoichiro Ueno, and Takashi Nanya. TITAC-2: An asynchronous 32-bit microprocessor based on scalable-delay-insensitive model. In *Proc. International Conf. Computer Design (ICCD)*, pages 288–294, October 1997.

[22] H. Saito. *Synthesis of Globally Delay Insensitive Locally Timed Asynchronous Circuits from Register Transfer Level Descriptions*. PhD thesis, University of Tokyo, 2003.

Appendix

Proof of Lemma 1 From $x \notin D$, the value of x distinguishes the elements in $\mathcal{C}_D(ES(x+))$ from those in $\mathcal{C}_D(QS(x+))$ and $\mathcal{C}_D(ES(x-))$. Similarly, $\mathcal{C}_D(QS(x-))$ is disjoint from $\mathcal{C}_D(QS(x+))$ and $\mathcal{C}_D(ES(x-))$. $\mathcal{C}_D(ES(x+))$ and $\mathcal{C}_D(QS(x-))$ are disjoint for the following reason. Suppose that they have a common element s. Then, there must exist $s_1 \in ES(x+)$ and $s_2 \in QS(x-)$ such that $s \in \mathcal{C}_D(s_1)$ and $s \in \mathcal{C}_D(s_2)$. Then, from the definition of D-closure, $\mathcal{C}_D(s_1) = \mathcal{C}_D(s_2)$ must hold, and so $s_2 \in \mathcal{C}_D(s_1)$ is implied. This, however, contradicts $\mathcal{C}_D(ES(x+)) - UR = ES(x+)$, because $ES(x+)$ and $QS(x-)$ are disjoint. Hence, $\mathcal{C}_D(ES(x+))$ and $\mathcal{C}_D(QS(x-))$ are disjoint. The disjointness between $\mathcal{C}_D(ES(x-))$ and $\mathcal{C}_D(QS(x+))$ can be shown similarly. (Q.E.D.)

Proof of Lemma 2 For this proof, we focus on the gC implementation, and furthermore, only $C(x+)$ is considered, because the proof for the other cover and the atomic gate implementation can be done similarly. Let ES' and QS' denote the excitation and stable state sets of G_D. From the construction of G_D, those are obtained by $ES'(x+) = \mathsf{proj}_D(\mathcal{C}_D(ES(x+)))$, $QS'(x+) = \mathsf{proj}_D(\mathcal{C}_D(QS(x+)))$, and so on. Since some unreachable states of G are mapped into those excitation or stable states of G_D, the unreachable state set UR' of G_D satisfies $\mathsf{proj}_D^{-1}(UR') \subseteq UR$.

The first thing to be shown is that G_D has CSC and is output semi-modular. The former is straightforward from Lemma 1. For the latter, suppose that G_D is not output semi-modular. This can happen if an output transition t_x is enabled by an irrelevant transition t_d in G, and t_d is replaced by a dummy transition in G_D, resulting in a new chain of dummy transitions to t_x from a dummy transition conflicting with some transition. Let $s_1 \xrightarrow{t_d} s_2$ be the state transition caused by t_d. Since t_d makes t_x enabled, $s_1 \notin ES(x+)$ and $s_2 \in ES(x+)$ hold (assuming that t_x is a rising transition of an output x). This, however, contradicts that t_d is a transition related to an irrelevant signal, because $s_1 \in \mathcal{C}_D(s_2)$ but $s_1 \notin ES(x+) \cup UR$. Therefore, G_D is output semi-modular. The similar discussion holds for the case that t_d is followed by a chain of dummy transitions. Since G is output semi-modular, there are no other cases that G_D is not output semi-modular. Hence, G_D can be concluded to be output semi-modular. The signals related to transitions like t_d are called trigger signals for x, and they do not belong to any irrelevant input set for x.

The above two facts guarantee the existence of a correct circuit with respect to G_D. Let $C'(x+)$ denote one of its covers. Note that this cover satisfies

$$ES'(x+) \subseteq C'(x+) - UR' \subseteq ES'(x+) \cup QS'(x+). \quad (1)$$

The next thing to be shown is that this cover also satisfies the correctness condition of G. Since $C'(x+)$ should be considered in the state space of G, this is shown by

$$ES(x+) \subseteq \mathsf{proj}_D^{-1}(C'(x+)) - UR \subseteq ES(x+) \cup QS(x+). \quad (2)$$

The above $(\mathsf{proj}_D^{-1}(C'(x+)) - UR)$ can be rewritten as follows.

$$\begin{aligned}
&\mathsf{proj}_D^{-1}(C'(x+)) - UR \\
&= \mathsf{proj}_D^{-1}((C'(x+) - UR') \cup (C'(x+) \cap UR')) - UR \\
&= \mathsf{proj}_D^{-1}(C'(x+) - UR') \cup \\
&\qquad \mathsf{proj}_D^{-1}(C'(x+) \cap UR') - UR \\
&= \mathsf{proj}_D^{-1}(C'(x+) - UR') - UR. \quad (3)
\end{aligned}$$

This (3) is obtained from $\mathsf{proj}_D^{-1}(UR') \subseteq UR$. Applying proj_D^{-1} to (1) derives

$$\begin{aligned}
&\mathsf{proj}_D^{-1}(ES'(x+)) \subseteq \mathsf{proj}_D^{-1}(C'(x+) - UR') \\
&\qquad \subseteq \mathsf{proj}_D^{-1}(ES'(x+) \cup QS'(x+)) \\
&\mathcal{C}_D(ES(x+)) \subseteq \mathsf{proj}_D^{-1}(C'(x+) - UR') \\
&\qquad \subseteq \mathcal{C}_D(ES(x+)) \cup \mathcal{C}_D(QS(x+)) \\
&\mathcal{C}_D(ES(x+)) - UR \subseteq \mathsf{proj}_D^{-1}(C'(x+) - UR') - UR \\
&\qquad \subseteq (\mathcal{C}_D(ES(x+)) - UR) \cup (\mathcal{C}_D(QS(x+)) - UR)) \\
&ES(x+) \subseteq \mathsf{proj}_D^{-1}(C'(x+) - UR') - UR \\
&\qquad \subseteq ES(x+) \cup QS(x+) \quad (4)
\end{aligned}$$

The final equation (4) holds because D is an irrelevant input set. From (3) and (4), (2) is obtained. (Q.E.D.)