Logic Optimization for Asynchronous Speed Independent Controllers using Transduction Method

Hiroshi Saito
RCAST
The University of Tokyo
Tokyo, JAPAN 153-8904
Tel: +81-3-5452-5160
Fax: +81-3-5452-5161
hiroshi@hal.rcast.u-tokyo.ac.jp

Hiroshi Nakamura
RCAST
The University of Tokyo
Tokyo, JAPAN 153-8904
Tel: +81-3-5452-5162
Fax: +81-3-5452-5163
nakamura@hal.rcast.u-tokyo.ac.jp

Masahiro Fujita
Dept. of EE
The University of Tokyo
Tokyo, JAPAN 113-8654
Tel: +81-3-5841-6764
Fax: +81-3-5841-6724
fujita@ee.t.u-tokyo.ac.jp

Takashi Nanya
RCAST
The University of Tokyo
Tokyo, JAPAN 153-8904
Tel: +81-3-5452-5160
Fax: +81-3-5452-5161
nanya@hal.rcast.u-tokyo.ac.jp

Abstract— Asynchronous speed independent (SI) circuits based on an unbounded gate delay model often suffer from high area penalty. It happens due to the lack of efficient global optimization. This paper presents a boolean optimization method based on tranduction method to optimize asynchronous SI circuits while preserving hazard-freeness.

I. Introduction

Signal Transition Graphs (STGs), interpreted Petri Nets, are commonly used to specify behaviors of asynchronous circuits. Starting from STGs, an asynchronous logic synthesis tool `petrify` [2] can synthesize the corresponding asynchronous speed-independent (SI) circuit, where the circuit behaves correctly under any gate delay.

Even though `petrify` is well established for the synthesis of asynchronous SI circuits, global optimizations using the relationships among logic functions of gates are not realized because each output function is derived separately from the encoded state graph of an STG. This sometimes causes redundant circuits such that the same function appears on the different logic networks.

To solve this problem, in this paper, we show an approach to optimize asynchronous SI circuits globally using don't cares derived from given circuit structure. The don't cares are exploited at the whole circuit structure by calculating *permissible functions*.

Permissible functions [4] are functions guaranteeing that a change within them does not affect the circuit outputs. They are frequently used at multi-level logic optimizations. One of the representative approaches using permissible functions is transduction method [4, 5]. The transduction method optimizes given circuits by sharing common gates and substituting gates so that the total number of gates and/or connections are minimized.

In this paper, we extend the transduction method to apply it for asynchronous SI controllers considering how to calculate permissible functions and which transformations guarantee hazard-freeness. In particular, we focus on *gate substitution algorithm* in [4]. Fig.1 shows the proposed optimization

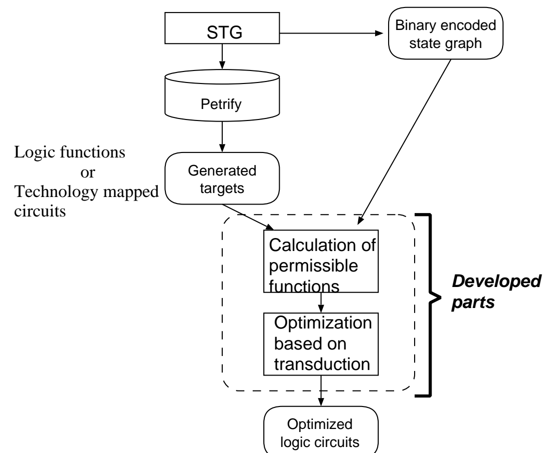

Fig. 1. Framework of this work

flow based on the transduction method. As initial inputs, it accepts logic functions for a circuit (or technology mapped circuit) produced by `petrify` and the corresponding binary encoded state graph. Then, it optimizes the circuit in terms of gate substitution if hazard-freeness is guaranteed.

According to [3], in `petrify`, global optimization in terms of gate substitution is carried out when a function is decomposed during technology mapping. However, it is restricted to whether some gate is substituted by the decomposed gate or not. In addition, the computation complexity is increased because recalculation of state space is required when each function is decomposed. Our approach does not restrict to some specific gate and impose recalculation of state space because it optimizes circuits globally without requiring any logic decomposition.

The rest of the paper is organized as follows. In section II, the basic notions of STG-based synthesis is presented. In section III, the calculation of permissible functions for asynchronous SI controllers is discussed. In section IV, we show how to substitute the gates in asynchronous SI circuits preserving hazard-freeness. Finally, we show the experimental results in section V and conclude this work in section VI.

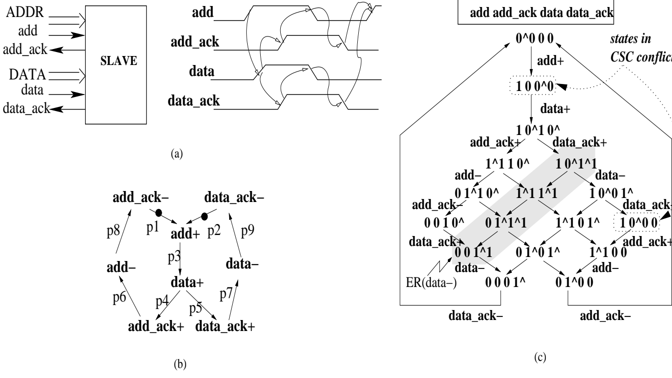

Fig. 2. Simple asynchronous interface: (a) timing diagrams, (b) STG, (c) SG

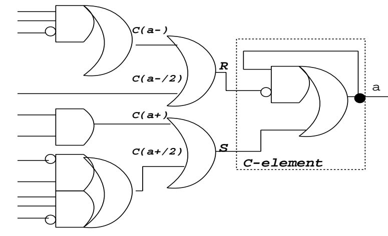

Fig. 3. gC-implementation

II. BACKGROUND

A. Signal Transition Graph

Fig.2.a shows a simple interface between two modules in an asynchronous system, a master (e.g., a processor) and a slave (e.g., memory). The interface involves two signal handshakes, one for controlling the transmission of address (add and add_{ack}) and the other for data ($data$ and $data_{ack}$). The timing diagram shown in Fig.2.a defines the synchronization protocol between the handshakes.

Fig.2.b shows the *Signal Transition Graph (*STG*)* [1] corresponding to the timing diagram of the controller. All nodes in the STG are interpreted as signal transitions: a rising transition of signal a is labeled with "$a+$" and a falling transition with "$a-$". We also use the notation $a*$ if we are not specific about the sign of the transition.

An STG transition is *enabled* if all its input places (arcs) contain a token. In the initial marking $\{p1, p2\}$ of the STG in Fig.2.b, transition $add+$ is enabled. Every enabled transition can fire, removing one token from every input place of the transition and adding one token to every output place. After the firing of transition $add+$ the token moves to a new marking, $\{p3\}$, where $data+$ is enabled.

The set of all signals in an STG is partitioned into a set of *inputs*, which come from the environment, and a set of *outputs* and *state signals* that must be implemented.

B. State Graph

Playing the token game for the reachability analysis on a given STG, one can generate a *State Graph (*SG*)* in which each node (a marking) is labeled with a vector of signal values (in Fig.2.c, signals that can change in the state are marked with "$\wedge$") and arcs between pairs of states are labeled with the corresponding fired transitions.

Excitation region and quiescent region. A maximally connected set of states in which $a*$ is enabled is called an *excitation region* (ER) for transition $a*$ (denoted by ER$(a*)$, e.g., the shadowed set of states in Fig.2.c corresponds to ER$(data-)$). The *quiescent region* (QR) for transition $a*$ (denoted by QR$(a*)$) is a maximal set of states such that a is stable and not reachable from any other ER$(a*$.

Signal consistency. An SG is *consistent* if in every transition sequence from the initial state, rising and falling transitions alternate for each signal. Fig.2.c shows the SG for the STG in Fig.2.b, which is consistent.

Implementability conditions. In addition to consistency, the following two properties are required for an SG to be implementable as a hazard-free asynchronous circuit. The first property is *output persistency*. A transition $a*$ is *persistent* in a state s if it is enabled in s and remains enabled in any other state reachable from s by executing another transition $b*$. An SG is *output persistent* if all output signal transitions are persistent in all states and input signals cannot be disabled by outputs.

The second implementability property, *Complete State Coding* (CSC), is necessary and sufficient for the existence of a logic circuit implementation. A consistent SG satisfies the CSC property if for every pair of states with the same binary codes the set of output transitions enabled in both states is the same. Pairs of states s, s' that violate the CSC condition are said to be in CSC *conflict* (binary codes 100*0 and 10*00 in Fig.2.c).

The following sufficient condition was proved in [1]: *an* STG *can be implemented by a speed-independent circuit if it is consistent, output-persistent, and* CSC.

C. Generalized C-element implementations

If previously discussed conditions are satisfied, one can produce an SI circuit out of an STG where each signal a will be implemented as $a = S + \overline{R} \cdot a$ form, where R and S are set and reset functions respectively. This way of implementation is known as generalized C-element implementation (or gC-implementation). In this work, we focus on the circuits derived by this implementation style as optimization targets.

Fig.3 shows an example of gC-implementation for a signal a. Each gate in the first level (i.e.,$C(a+), C(a + /2)$) corresponds to a signal transition and is derived to satisfy the following *monotonous cover conditions*. Note $a * /i$ means the i-th transition of signal transition $a*$.

1. Cover condition: $C(a * /i)$ covers all states of ER$(a * /i)$ (i.e., $C(a * /i)$ evaluates to 1 in all states of ER$(a * /i)$)

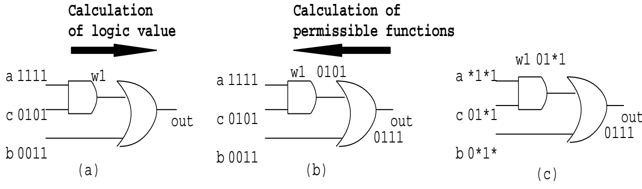

Fig. 4. Calculation of permissible functions

2. One-hot condition: $C(a*/i)$ does not cover any state outside $\mathsf{ER}(a*/i) \cup \mathsf{QR}(a*/i)$
3. Monotonicity condition: $C(a*/i)$ changes at most once along any state sequence within $\mathsf{QR}(a*/i)$

The same signal transitions with different instance i are gathered in the second level with an OR gate. It represents either set or reset function.

III. Calculation of Permissible Functions

A. Permissible Functions

Permissible functions, defined for each net and gate output, represent a set of logic functions in which a change within the functions does not affect circuit outputs (due to don't care space derived from circuit structures).

According to [4], the calculation of permissible functions consists of the following two steps.

1. Calculation of logic values for each gate and net by assigning truth values for the primary inputs (Fig.4.b)
2. Calculation of permissible functions for each logic and net starting from the circuit outputs to the primary inputs (Fig.4.c)

After all of the logic values are calculated (Fig.4.b), the permissible functions are derived by assigning possible don't care for each input. For example in an OR gate, all of the inputs must be 0 if the output is equal to 0. However, if the output is equal to 1, one of its inputs must be 1 while the others can be either 0 or 1 (i.e., don't care, denoted by $*$). In Fig.4.b, the permissible functions of the inputs of the OR gate, $w1$ and b, can be 01*1 and 0*1*. Following to the same consideration for the AND gate, we can obtain the permissible functions of the circuit in Fig.4.a as in Fig.4.c.

In some cases, a gate (or a net) may have several candidates of the permissible functions. For example in Fig.4.c, there is another possibility of the permissible functions for $w1$ and b, 01** and 0*11 (the last element is different from the previous case). The difference comes from the choices of the don't care assignments for $w1$ and b where the output of the OR gate is 1. In fact, since calculations of all candidates require lots of computation time, we concentrate on only a partial set of permissible functions called *Compatible Set of Permissible Functions (*CSPF*)*.

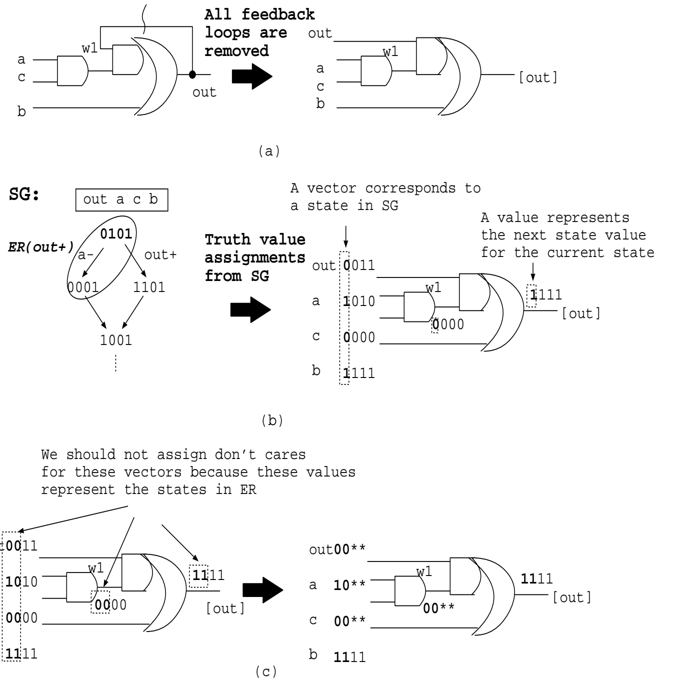

Fig. 5. Calculation of permissible functions for asynchronous SI circuits

B. Calculation of Permissible Functions for Asynchronous SI Circuits

In addition to the previous procedures, the following considerations are required for the calculation of permissible functions in asynchronous SI controllers.

1. Removal of all feedback loops
2. Assignments of truth values from corresponding SG
3. Assignments of don't care except the states in ERs

Looking through asynchronous SI circuits, they contain *feedback loops* because they describe sequential machines. To prevent the iterative calculations caused by these loops, we must cut all loops as in Fig.5.a before the calculation of logic values.

In addition, since the behaviors of asynchronous SI controllers are represented by the states in the corresponding SGs, the truth values of signals directly come from the corresponding SGs (see Fig.5.b).

The last requirement is don't care assignment. In asynchronous SI circuits, since the timing of the signal changes of non-input signals (i.e, output or state signals) is represented as an ER on SG, assignments of don't care to the states in ERs may lead to some hazardous behavior during transformations. Therefore, we do not assign don't care for any state in ER. It must be considered on all of the gates and the nets in a given circuit. Fig.5.c shows the calculation result of the permissible functions for Fig.5.a.

IV. Transduction Method for Asynchronous SI Circuits

A. Validations of gate substitutions

In order to preserve hazard-freeness after optimizations, our approach allows substitution if gate $g2$ which substitutes gate $g1$ satisfies the monotonous cover conditions of the gate $g1$. This is checked by observing the relationships of the logic values and the corresponding ERs and QRs for both $g1$ and $g2$. Before describing formal substitution conditions in gC-implementations, we define several terminologies.

- $G_s(g)$ - the value of CSPF of gate g in state s
- $pred_s$ - set of immediate predecessor states for state s

Proposition IV.1 *The substitution of gate $g1$ by gate $g2$ is hazard-free if the following conditions are satisfied.*

1. $\forall$ state $s : G_s(g1) = 1 \Rightarrow s \in g2$ (i.e., $g2$ evaluates to 1 in state s)
2. $\forall$ state $s : s \notin \mathsf{ER}(a*/i) \cup \mathsf{QR}(a*/i) \Rightarrow s \notin g2$
3. $\forall$ state $s : s \in \mathsf{QR}(a*/i) \cap g2 \Rightarrow pred_s \in g2$

Note $g1$ is a gate for the i-th transition of signal transition $a*$.

Proof of proposition IV.1. The first condition in Prop.IV.1 guarantees the cover condition in the monotonous cover conditions. Since we have never assigned don't care for all of the states in ER, $G_s(g1)$ is equal to 1 if s is a state in $\mathsf{ER}(a*/i)$. Under such a situation, if $g2$ does not cover the state s (i.e., $g2 = 0$ in state s), it looses the timing to produce signal transition $a*/i$ which implies a hazardous behavior.

The second condition is for the one-hot condition in the monotonous cover conditions. According to the one-hot condition, gate $g1$ does not cover the states out of $\mathsf{ER}(a*/i) \cup \mathsf{QR}(a*/i)$. If gate $g2$ covers such states and substitutes $g1$, it means that there exist hazardous behaviors for gate $g2$ in those states, which may be propagated to the circuit outputs.

The third condition is for the monotonicity condition in the monotonous cover conditions. $s \in \mathsf{QR}(a*/i) \cap g2$ means the state which is in $\mathsf{QR}(a*/i)$ and $g2$ evaluates to 1. In such a state, if one of the immediate predecessor state s' (i.e., s' is a state in $pred_s$) is not covered by $g2$, there is an additional transition of $g2$ between s' and s ($g2$ is 0 in s' but 1 in s), which violates the monotonicity condition of signal transition $a*/i$. □

Before the transformations, Prop.IV.1 is checked to validate hazard-freeness. If one of them is not satisfied, the substitution is prevented because it may lead to any hazardous behavior.

B. Gate Substitution Algorithm

As a transduction method, we focus on *gate substitution algorithm* in [4], while extending it to satisfy Prop.IV.1. The gate substitution algorithm allows substitution of gate $g1$ by gate $g2$ if in their CSPFs, $G(g1)$ and $G(g2)$, $G(g1)$ includes $G(g2)$ ($G(g1) \supset G(g2)$). For example in Fig.6, $g1$ is substituted by $g2$ because $G(g1) = 01** \supset G(g2) = 011*$.

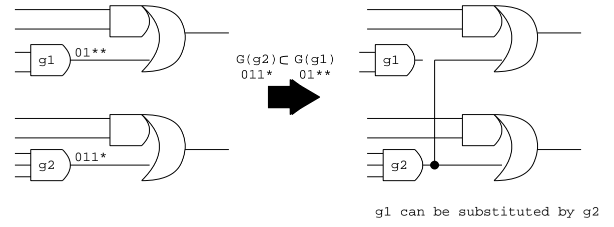

Fig. 6. An illustration of gate substitution

In our extended gate substitution algorithm, the transformations are classified by the statements of calculated CSPFs and logic values.

Case1: The logic values of both gates are equivalent. If in all states the logic values of two gates $g1$ and $g2$ are equivalent, $g1$ is substituted by $g2$ without caring anything because in all states they have the same value. Substitution of $g2$ by $g1$ is also possible.

Case2: $G(g2) \subseteq G(g1)$ (or $G(g1) \subset G(g2)$). Since $G(g2) \subseteq G(g1)$ means that in all states where $G_s(g1)$ is constant 0 or 1 $G_s(g2)$ has the same value, the first condition of Prop.IV.1 is satisfied. This is because in all states of an $\mathsf{ER}(a*/i)$ where $G_s(g1)$ is 1, $g2$ also evaluates to 1. However, all the other conditions of Prop.IV.1 must be checked before transformations. We call the check of Prop.IV.1 in Case 2 as *Case2_check*.

Case3: Other cases. In all other cases, we calculate the conjunction of $G(g1)$ and $G(g2)$ ($NewG$ in Fig.7). If the conjunction is not empty, we check whether $g1$ or $g2$ is included in that conjunction or not ($NewG \supseteq g1$ or $NewG \supseteq g2$). When $g1$ is included in that conjunction, $g2$ is substituted by $g1$ if for $g1$ all of the conditions in Prop.IV.1 with respect to $g2$ are satisfied. We call this check as *Case3_check*.

If the conjunction of CSPFs exists but $g1$ or $g2$ is not included, we create a set of new gates (New in Fig.7) such that each new gate g is included in the conjunction ($NewG \supseteq g$). In this case, we must check all of the conditions of Prop. IV.1 for the newly created gate with respect to $g1$ and $g2$.

Fig.7 shows a pseudo code of the extended gate substitution algorithm.

Example. In order to demonstrate how the extended gate substitution algorithm works, we apply it for an example circuit. Fig.8.a shows the SG of this example and Fig.8.b shows a part of the corresponding SI circuit with respect to gC-implementation. The CSPFs are assigned for C-element, set, and reset functions (i.e., C-elements for $aout$ and csc, gate $g1$, and gate $g2$). Since CSPFs of $g2$ and $g3$ are neither $G(g2) \subseteq G(g3)$ nor $G(g2) \supset G(g3)$ and the conjunction of them is not empty ($G(g2) \cap G(g3)$ = 11*100000000000000), this is Case3.

Suppose we substitute gate $g3$ by gate $g2$. For the first condition of Prop.IV.1, $Case3_check$ checks the states where

Input: An initial SI circuit and the corresponding SG
Output: An optimized SI circuit

```
begin
calculate CSPF G(gi) for all i using SG
while network is changed do
  foreach pair of gates, gi and gj (i ≠ j) do
    /* Case1 */
    if ∀ state s:  g1 = g2
      disconnect all connection to gj
      connect gi to all fanout gates of gj
      recalculate CSPF G(gi) for all i
      break
    endif
    /* Case2 */
    if G(gi) ⊇ G(gj) and Case2_check of gi wrt gj
      disconnect all connection to gi
      connect gj to all fanout gates of gi
      recalculate CSPF G(gi) for all i
      break
    endif
    if G(gj) ⊃ G(gi) and Case2_check of gj wrt gi
      disconnect all connection to gj
      connect gi to all fanout gates of gj
      recalculate CSPF G(gi) for all i
      break
    endif
    /* Case 3 */
    NewG = G(gi) ∩ G(gj) /* conjunction of G(gi) and G(gj) */
    if NewG ≠ ∅
      if NewG ⊇ gi and Case3_check of gi wrt gj
        disconnect all connection to gj
        connect gi to all fanout gates of gj
        recalculate CSPF G(gi) for all i
        break
      endif
      if NewG ⊇ gj and Case3_check of gj wrt gi
        disconnect all connection to gi
        connect gj to all fanout gates of gi
        recalculate CSPF G(gi) for all i
        break
      endif
      New = {g|g ∈ {gi}, g is connectable to NewG}
      if NewG ⊃ g in New and Case3_check of g wrt gi and gj
        disconnect all connection to gi, gj
        connect g to all fanout gates of gi, gj
        recalculate CSPF G(gi) for all i
        break
      endif
    endif
  endforeach
endwhile
end
```

Fig. 7. Extended gate substitution algorithm

$G_s(g3)$ is 1. States $s1$ and $s2$ correspond to such states. Since the logic values of $g2$ in those states are 1, the first condition of Prop.IV.1 is satisfied ($s1$ and $s2$ are states in ER$(csc-)$ covered by $g3$, hence satisfying this condition means satisfying the cover condition in the monotonous cover conditions). For the second condition, states out of ER$(csc-) \cup$ QR$(csc-)$ are enumerated. Through the SG, states $s6$ - $s10$ and $s15$ - $s18$ correspond to such states. In all of those states, since $g2$ does not cover those states (evaluates to 0), the second condition of Prop.IV.1 is also satisfied. For the last condition of Prop.IV.1, states in QR$(csc-)$ and covered by $g2$ are enumerated. States $s3$ and $s4$ correspond to such states. In such states, the last condition checks whether all of the immediate predecessor states of $s3$ and $s4$ are covered by $g2$ or not. In states $s1$, $s2$, and $s4$ ($s1$ and $s4$ are immediate predecessor states of $s3$ and $s2$ is

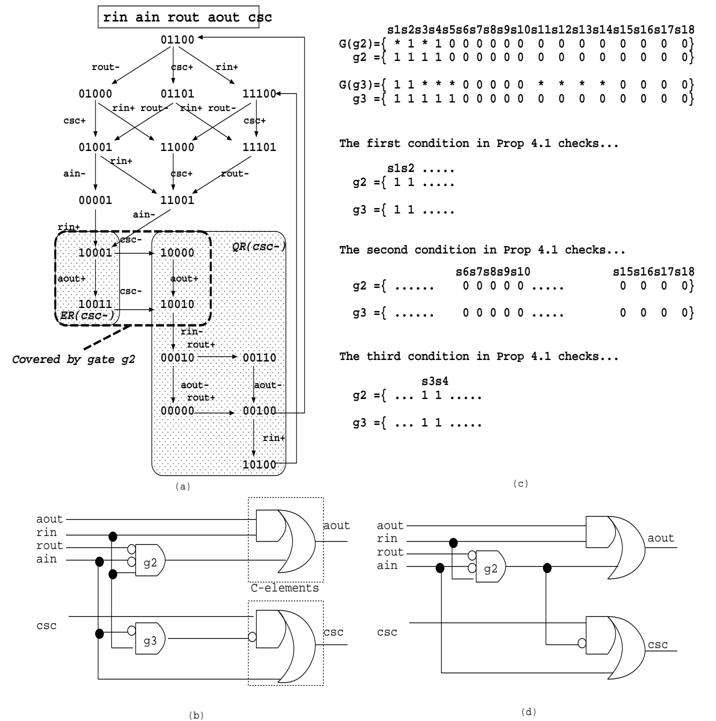

Fig. 8. An example of gate substitutions

of $s4$), since $g2$ covers those states, the last condition is also satisfied. As a result, the substitution of $g3$ by $g2$ is carried out without leading any hazardous behavior (Fig.8.d).

On the other hand, suppose we substitute $g2$ by $g3$. In this case, we can see a violation of the second condition of Prop IV.1 in state s5 (11101). In state $s5$ which is outside of ER$(aout+) \cup$ QR$(aout+)$, $g3$ covers $s5$. This means that $g3$ has a $0 \rightarrow 1 \rightarrow 0$ hazard for $aout+$ around state $s5$ if $g2$ is substituted by $g3$. Therefore our extended gate substitution algorithm prevents the substitution.

V. Experimental Results

In this section, we show the experimental results of logic optimizations applying our gate substitution algorithm to gC-implementations and technology mapped circuits. For this, we implemented the extended gate substitution algorithm using JAVA. The experimental environment is Windows 98 with a Pentium II processor (300MHz) and a 64 M byte memory.

Note, in technology mapped circuits, the monotonous cover conditions for each gate are little bit different from gC-implementations (see [3]) due to the introduction of decomposed gates. Although the formal conditions for substitutions must be considered, we will investigate the applicability of our gate substitution algorithm for technology mapped circuits. Hence, in this work, the functionalities and hazard-freeness of optimized circuits were verified by using SI verification tool versify. The formal conditions for substitutions will be considered in our future work.

TABLE I
RESULTS OF OPTIMIZATIONS IN gC-IMPLEMENTATIONS

name	states	Original SI		Optimized SI		time
		nodes	lits.	nodes	lits.	(sec.)
nak-pa	58	15	37	10	26	10.11
fifo	18	6	17	5	15	1.10
mp-for	22	11	29	9	25	2.30
ram-read	39	13	35	12	32	4.06

TABLE II
RESULTS OF OPTIMIZATIONS IN TECHNOLOGY MAPPED CIRCUITS

name	states	Original SI		Optimized SI		time
		nodes	lits.	nodes	lits.	(sec.)
nak-pa	58	13	26	12	24	3.51
fifo	18	8	22	7	21	1.04
converta	24	15	36	13	34	5.39
mul	47	9	22	7	20	6.04
sbuf	81	13	31	12	30	6.54

A. Experiments on gC-Implementations

In this experiment, CSPFs are assigned for C-element, set, and reset functions. Table I shows the optimization results for gC-implementations of benchmark circuits. The second column shows the number of states in SGs. The third and forth ones show the number of nodes and literals in the original SI circuits. The fifth and sixth ones show the results after the optimizations respectively. The final column shows the calculation time.

The result shows that we can reduce 20% of the area with respect to the number of nodes (17% wrt the number of literals) on average. In gC-implementations, our approach works well while substituting identical gates or the gates which have similar logics.

B. Experiments on Technology Mapped Circuits

Similar to the previous experiments, we apply our gate substitution algorithm to technology mapped benchmark circuits. In this experiment, we assume that our library has C-element ($c = a \cdot b + (a+b) \cdot c$), AND, OR, NAND, NOR and INV gates under three fanin. CSPFs are assigned for each gate.

Table II shows the result of this experiment. From the result, the area reduction is about 10% wrt the number of nodes (6% wrt the num. of lits.) on average. In this experiment the effect is not so much compared to gC-implementations because `petrify` tries gate substitution when a function is decomposed. However, our approach can optimize the circuits even after the gate substitution is carried out.

VI. CONCLUSION

Because of the lack of efficient global optimizations in asynchronous SI circuit synthesis, the resulting circuits sometimes have redundant circuitry. To solve this problem, we proposed an optimization method for asynchronous SI controllers globally using transduction method while extending it to preserve hazard-freeness. The experimental results were encouraging in that on average the area reductions by our approach were about 20% (for gC-implementations) and 10% (for technology mapped circuits) in terms of the number of nodes. The algorithm discussed in this paper was implemented using JAVA.

For future works, another optimization method based on maximal set of permissible functions is considered because it may give better results. In addition, the formal substitution conditions for technology mapped circuits will be considered.

ACKNOWLEDGEMENTS

We would like to thank Dr. Alex Kondratyev (Cadence Barkley Lab.) and Prof. Jordi Cortadella (Universitat Politecnica de Catalunya) for their useful comments in this paper. This work is supported by International-Technology Promotion Agency (IPA) in Japan.

REFERENCES

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

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

[3] A. Kondratyev, M. Kishinevsky, J. Cortadella, L. Lavagno, and A. Yakovlev. Technology mapping for speed-independent circuits: decomposition and resynthesis. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 240–253. IEEE Computer Society Press, April 1997.

[4] S. Muroga, Y. Kambayashi, H.C. Lai, and J.N. Culliney. The transduction method - design of logic networks based on permissible functions. *IEEE Transactions on Computers*, 1989.

[5] M. Futita. A logic synthesis system with multi-level logic circuit minimization mechanism based on transduction methods. *IPSJ transaction*, May 1989.

[6] A. Kondratyev, M. Kishinevsky, B. Lin, P. Vanbekbergen, and A. Yakovlev. Basic Gate Implementation of Speed-Independent Circuits. In *Proc. Design Automation Conference*, pages 56–62, June 1994.