システムＬＳＩ設計技術　106－3
（２００２．５．２３）

# トランスダクション法を用いた非同期制御回路最適化

齋藤　寛[†]　中村　宏[†]　藤田　昌宏[††]　南谷　崇[†]

[†] 東京大学先端科学技術研究センター　〒 153–8904 東京都目黒区駒場 4–6–1
[††] 東京大学工学部電子工学専攻　〒 113–8654 東京都文京区本郷 7–3–1
E-mail: †{hiroshi,nakamura,nanya}@hal.rcast.u-tokyo.ac.jp, ††fujita@ee.t.u-tokyo.ac.jp

**あらまし**　微細化による配線遅延の影響の為に、単一クロックを用いて回路全体を制御することは困難になりつつある。非同期回路はグローバルクロックを用いずに回路を制御するため、このような問題の解決法の一つになると考えられる。またクロックがないことより、低消費電力、低電磁波といった利点が挙げられる。その一方、unbounded ゲート遅延モデルである非同期 speed independent（SI）回路は、ハザードフリーといった要求のため、回路規模が同期のものと比べて大きくなる傾向がある。従って本報告では、非同期 SI 回路の面積最適化の為、従来の組み合わせ回路最適化で用いられるトランスダクション法を拡張し、JAVA を用いて実装を行なった。

**キーワード**　非同期 SI 回路, 許容関数, トランスダクション法, ハザード

# Logic Optimization for Asynchronous SI Controllers using Transduction Method

Hiroshi SAITO[†], Hiroshi NAKAMURA[†], Masahiro FUJITA[††], and Takashi NANYA[†]

[†] Faculty of Research Center for Advanced Science and Technology, University of Tokyo
Komaba 4–6–1, Meguro-ku, Tokyo, 153–8904 Japan
[††] Faculty of Electronics Enginnering, University of Tokyo
Hongo 7–3–1, Bunkyo-ku, Tokyo, 113–8654 Japan
E-mail: †{hiroshi,nakamura,nanya}@hal.rcast.u-tokyo.ac.jp, ††fujita@ee.t.u-tokyo.ac.jp

**Abstract**　Due to wire delays caused by shrinking down of circuit components, the control of a circuit using a global clock has become more difficult. Since asynchronous circuits do not require such a global clock, they will be the good candidate to solve this problem. In addition, due to the lack of the global clock, asynchronous circuits offer a number of potential advantages such as low power consumption and electro-magnetic compatibility. On the other hand, because of the requirement of hazard-freeness, asynchronous speed independent (SI) circuit, which is an unbounded gate model, sometimes suffers from high area penalty more than synchronous one. Therefore, in order to reduce area overhead, transduction method, a well used optimization method in multi-level logic, is extended for hazard-free asynchronous SI circuits. In addition, we implemented it using JAVA.

**Key words**　Asynchronous SI Circuits, Permissible Functions, Transduction Method, Hazards

## 1. Introduction

Since wire delay has become more dominant over gate delay in deep sub-micron technologies (DSM), the control of a circuit using a global clock is getting difficult. Asynchronous circuits, free from such a global clock, become a potential solution for this problem. In addition, they offer a number of potential advantages such as reduced risk of synchronization failures, low power consumption, improved noise and electro-magnetic compatibility to name but a few.

Signal Transition Graphs (STGs), interpreted Petri Nets, are one of the well used specifications to describe asynchronous behaviors. Starting from this specification, asynchronous logic synthesis tool `petrify` [2] synthesizes the corresponding asynchronous speed-independent (SI) circuit, where the circuit behaviors are correctly operated under any gate delay.

Even though `petrify` is well established for the synthesis of asynchronous SI circuits, global optimizations using the relation of logic functions are not realized because each output function is derived separately from the encoded state graph. This sometimes causes redundant circuit such that some function appears on the different logic networks. In addition, hazard-freeness in asynchronous circuits also poses area overhead for the resulting circuit. From these points of views, the optimization is one of the important issues in the synthesis of asynchronous circuits.

To answer this question, in this paper, we optimize asynchronous SI circuits globally, using don't care space derived from given circuit structures. The don't care is exploited at the whole circuit structure by calculating *permissible functions*.

Permissible functions [6] are functions guaranteeing that a change in a given circuit does not affect the circuit outputs. They are well used for traditional multi-level combinational circuit optimizations. One of the representative approaches using permissible functions is transduction method [4], [6]. The transduction method optimizes the circuit by sharing common gates, substituting gates, and operating the routing while making redundancy through putting wires and removing them such that the resulting circuit is optimized.

In this paper, we extend the transduction method for asynchronous SI controllers. To realize it, we propose a framework of our optimization method including how to calculate permissible functions and which transformations are acceptable for hazard-free asynchronous SI circuits. In fact, we focus on only asynchronous SI controllers because the data path part will be handled by using traditional approaches applied for the multi-level combinational circuits.

Fig.1 shows the proposed optimization flow based on the

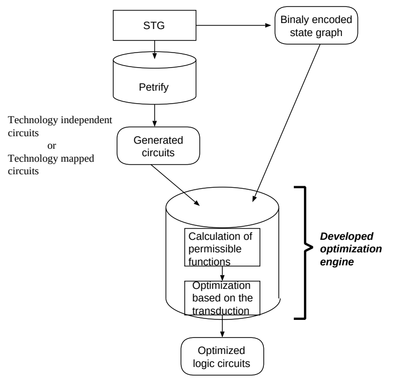

STG
Binaly encoded state graph
Petrify
Technology independent circuits or Technology mapped circuits
Generated circuits
Calculation of permissible functions
Optimization based on the transduction
Developed optimization engine
Optimized logic circuits

Fig. 1 Framework of this work

transduction method. As an initial circuit, the logic functions derived from an STG, are used. For such a given logic circuit, we calculate permissible functions. After the calculation of permissible functions, the circuit is optimized by the transduction method while preserving hazard-freeness using the notions of STG-based synthesis.

In the rest of the paper, we firstly show the basic notions of STG-based synthesis (in section 2.). In section 3., the calculation of permissible functions in asynchronous SI controllers is discussed. In section 4., we show how to transform the circuit preserving hazard-free conditions. Finally, we show the experimental results in section 5. and conclude this work in section 6..

## 2. Background

### 2.1 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 an 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 for the case of writing data into the slave.

Fig.2.b shows the Petri Net (PN) corresponding to the timing diagram of the controller. All events in this PN 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. PNs with such an interpretation are called *Signal Transition Graphs (or* STG*s)* [1] (Fig.2.c). STGs are typically represented in a "shorthand" form, where places with one input and one output arc are implicit.

An STG transition is *enabled* if all its input places contain a token. In the initial marking $\{p1, p2\}$ of the STG in Fig.2.c, transition $add+$ is enabled. Every enabled transition can fire, removing one token from every input place of the transition

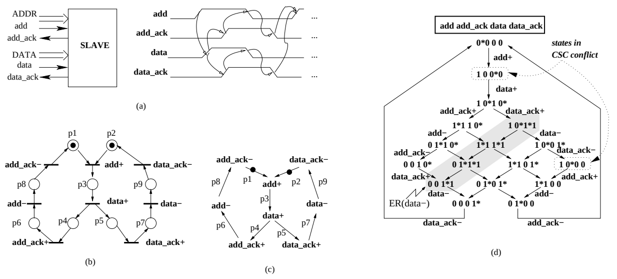


Fig. 2 Simple asynchronous interface: (a) timing diagrams, (b) PN, (c) STG, (d) SG

and adding one token to every output place. After the firing of transition *add*+ the token moves to a new marking, $\{p3\}$, where *data*+ becomes 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 *output* and *state* signals that must be implemented.

**2. 2 State Graph**

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

**Excitation region and quiescent region.** A maximally connected set of states in which $a*$ is enabled is called an *excitation region* (ER) for event $a*$ (denoted by ER($a*$), e.g. the shadowed set of states in Fig.2.d corresponds to ER(*data*−)). Excitation regions in an SG correspond to transitions in the STG. $b*$ is called a *trigger event* wrt $a*$ if in SG firing $b*$ from some state $s$ enters excitation region ER($a*$). In an STG triggering $a*$ by $b*$ implies the existence of direct causal relations between $a*$ and $b*$, i.e., either $b* \rightarrow a*$ or they are mediated by a place. All other states not in ER($a*$) are stable states for $a$ which is called *quiescent region* (denoted as QR($a*$)).

**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.d shows the SG for the STG in Fig.2.c, 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 *speed independence* which reduces to output-persistency of SG events. An event $a*$ is *persistent* in a state $s$ if it is enabled in $s$ and remains enabled in any other state reachable from $s$ by firing another event $b*$. An SG is *output-persistent* if all output signal events 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 events 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.d).

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

**2. 3 Target Circuit Structure**

If previously discussed conditions are satisfied one can produce an SI circuit out of an STG in which each signal $a$ will be implemented in $a = S + \overline{R} * a$ form, where $R$ and $S$ are set and reset gate functions respectively. This way of implementation is known as generalized C-element implementation (or gC-implementation simply).

## 3. Calculation of Permissible Functions

**3. 1 Permissible Functions**

Permissible functions, defined for each net and gate output, represent a set of logic functions in which a change on the functions does not affect the circuit outputs. It means that permissible functions reflect don't care space for the circuit structure. Based on permissible functions we can optimize the circuit.

According to [6], 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.3.a and b)

（2） Calculation of permissible functions for each logic and net starting from circuit outputs to the primary inputs (Fig.3.b and c)

After all of the logic values are calculated (Fig.3.b), the permissible functions are derived assigning possible don't care for each logic value. 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 others can be either 0 or 1 (i.e., don't care, denoted by $*$). For example in Fig.3.b, the permissible functions of the inputs of OR gate,$w1$ and $b$, are 00**01*1 and 00110*1*. Following to the same consideration for AND gate, we can get the permissible functions of Fig.3.a respectively (see Fig.3.c).

In some cases, a gate (or a net) has several candidates of the permissible functions. For example in Fig.3.c, there is another possibility of the permissible functions for $w1$ and $b$. I.e., the last element of $w1$ can be $*$ while the last one of $b$ is 1 (00**01** and 00110*11). To calculate all of the candidates requires much computation time with bigger memory space, therefore we concentrate on only a partial set of permissible functions called *Compatible Set of Permissible Function*

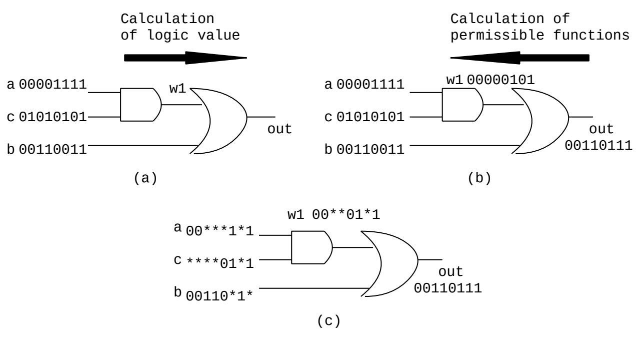


Fig. 3 Calculation of permissible functions

(CSPF).

### 3.2 Calculation of Permissible Functions for Asynchronous SI Circuits

In addition to the previous procedures, three requirements exist for the calculation of permissible functions in asynchronous SI controllers.

(1) Cut off all feedback loops

(2) Assignments of truth values from the corresponding SG

(3) Assignments of don't care except ERs

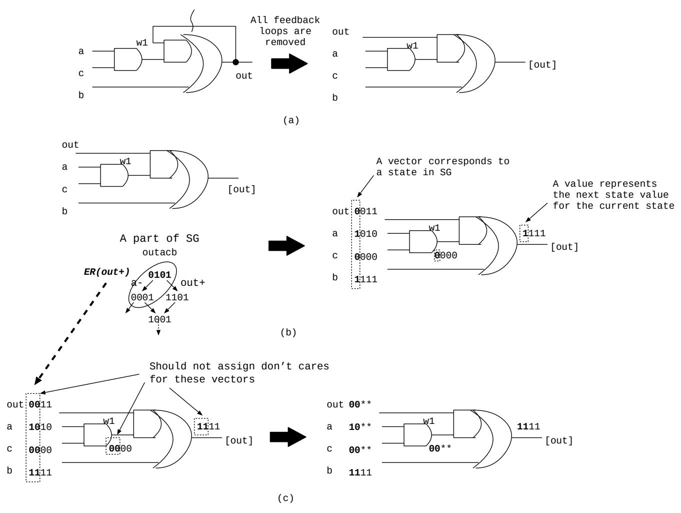
Looking through an asynchronous SI circuit, it contains *feedback loops* because it describes a sequential machine. To prevent the iterative calculation caused by these loops, we cut all of such loops as in Fig.4.a before the calculation of logic values.

Due to the nature of sequential machine, the logic values on the outputs represent the next state value corresponding to the current state (i.e., a vector of signal values corresponds to a state in SG). Therefore, the truth values of the signals directly come from the corresponding SG (see Fig.4.b).

The last requirement exists in the don't care assignment. In asynchronous SI circuits, a timing of signal changes in a non-input signal (i.e, an output or a state signal) is represented as ER on SG. Since an ER is configured from states satisfying *speed independent preserving sets* which are necessary and sufficient for hazard-free asynchronous SI circuits [3], assignments of don't care to ERs may lead some hazardous behavior. Therefore, to preserve this timing condition, we should not assign don't care value for any state in ER. This must be considered not only gates which are immediately followed by output signals but also all of the internal gates and nets. Fig.4.c shows the calculation result of permissible functions.

## 4. Transduction Method for Asynchronous SI Circuits

In this paper, we focus on an algorithm shown in [4], called *gate merge* algorithm, while extending it for hazard-free asynchronous SI controllers.



All feedback loops are removed
(a)
A vector corresponds to a state in SG
A value represents the next state value for the current state
A part of SG
outacb
ER(out+)
0101
a-
out+
0001
1101
1001
(b)
Should not assign don't cares for these vectors
[out]
(c)

Fig. 4 Calculation of permissible functions for asynchronous SI circuit

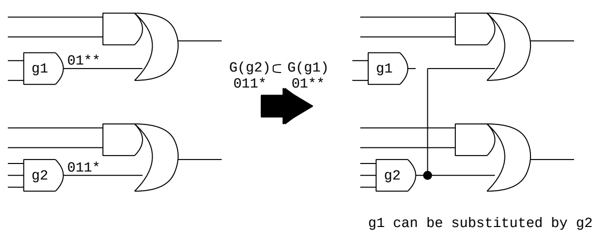


Fig. 5 An illustration of gate merge

The gate merge algorithm allows sharing of two gates $g1$ and $g2$ if the conjunction of their CSPFs $G(g1)$ and $G(g2)$ is not *empty* and one includes the other. For instance, if $G(g1) \supset G(g2)$, $g1$ is substituted by $g2$ because $G(g2)$ is included in $G(g1)$ (see Fig. 5).

According to [7], a gate $g(a_i*)$ in asynchronous SI controllers is generated such that the following three conditions are satisfied (such a gate is called *monotonous cover*). Note $g(a_i*)$ corresponds to the i-th transition of signal $a$.

- $g(a_i*)$ covers all states of ER$(a_i*)$
- $g(a_i*)$ changes at most once in any trace of states inside ER$(a_i*) \cup$ QR$(a_i*)$
- $g(a_i*)$ does not cover any reachable state outside ER$(a_i*) \cup$ QR$(a_i*)$

In order to prevent erroneous transformations, we must check the relations of these conditions and the calculated CSPFs. Since the calculated CSPFs have don't care assignments, the above conditions are modified as follows (suppose $g1$ is substituted by $g2$ here).

(1) $g2$ covers all states of ER$(a_i*)$ covered by $g1$

(2) $g2$ changes at most once in any trace of states inside ER$(a_i*) \cup$ QR$(a_i*)$ covered by $g1$

(3) $g2$ does not cover any reachable state outside ER$(a_i*) \cup$ QR$(a_i*)$ covered by $g1$ if such a state is not don't care

（4） $g2$ has the same logic value in all states where $g1$ is constant and $g2$ is don't care

Conditions (1) and (2) are equivalent to the monotonous cover conditions, however, condition (3) is somewhat relaxed. There may exist a state which is don't care for $g1$ but covered by $g2$, outside $\mathsf{ER}(a_i*) \cup \mathsf{QR}(a_i*)$. Although it violates the monotonous cover condition, there is no hazardous behavior after the substitution because in such a state $g1$ is don't care. Condition (4) is additional one. If $g1$ is constant and $g2$ is don't care in a state, there is a possibility of hazardous behavior if the logic value of $g2$ has opposit constant value. From this reason, the condition (4) is required.

These conditions must be checked before the transformations. The check itself is classified into three cases based on the relations of logic values and CSPFs between gates.

**Case1: The logic values of both gates are equivalent.** If logic values of two gates $g1$ and $g2$ are equivalent, then we can merge both gates easily without caring anything because in all states they has the same value. Hence, all conditions are satisfied.

**Case2: $G(g2) \subseteqq G(g1)$ (or $G(g1) \subset G(g2)$).** In the calculation of CSPFs, since we have never assigned don't care for ERs, these values are remained as constant. $G(g2) \subseteqq G(g1)$ means that all states where $G(g1)$ is constant is also the same constant in $G(g2)$, hence $g2$ covers all ERs covered by $g1$ (i.e., (1) and (4) are satisfied). Condition (3) is also satisfied because even if there is a state outside of $\mathsf{ER}(a_i*) \cup \mathsf{QR}(a_i*)$, it is don't care for $g1$, otherwise $G(g2) \subseteqq G(g1)$ is not satisfied, For condition (2), we must check the states where $G(g1)$ are constants but in QR and $G(g2)$ with constants but in ER. Under such a situation, the substitution of $g1$ by $g2$ may lead another hazardous behavior because the state which is stable for $g1$ will be enabled by $g2$, which implies a new transition within $\mathsf{ER}(a_i*) \cup \mathsf{QR}(a_i*)$. This is a violating case of condition (2), hence we must prevent the substitution if $G(g1)$ and $G(g2)$ has such a relation.

In Case2, only the check of condition (2) is required before the transformation. We call this check as *Case2_check*.

**Case3: Other cases.** In all other cases we must calculate the conjunction of $G(g1)$ and $G(g2)$. If the conjunction is not empty (i.e., condition (3) is satisfied, otherwise the conjunction will be empty), we check whether the conjunction is included by one of the CSPFs. Different from Case2, all of the conditions except condition (3) must be checked. We call this check as *Case3_check*. If the conjunction of CSPFs exists but not covered by one of them, we create a new gate if its logic values are covered by that conjunction. Of course, we must check conditions (1), (2), and (4) for the newly created gate with respect to $g1$ and $g2$.

Including these considerations, the extended gate merge algorithm for asynchronous SI controllers is obtained (see Fig.6).

```
Input:  An SI circuit and the corresponding SG
Output:  An SI circuit after gate merge

begin
calculate CSPF $G(gi)$ for all $i$ using SG
while network is changed do
  foreach $gi$, $gj$ of gates and $i > j$ do
    if logic values of $gi$ and $gj$, $F(gi)$ and $F(gj)$, are the same /* case 1 */
      disconnect all connection to $gj$ and connect $gi$ to all fanout gates of $gj$
      recalculate CSPF $G(gi)$ for all $i$
      break
    endif
    if $G(gi) \supseteqq G(gj)$ and $Case2\_check$ in $gi$ wrt $gj$ /* case 2 */
      disconnect all connection to $gi$ and connect $gj$ to all fanout gates of $gi$
      recalculate CSPF $G(gi)$ for all $i$
      break
    endif
    if $G(gj) \supset G(gi)$ and $Case2\_check$ in $gj$ wrt $gi$ /* case 2 */
      disconnect all connection to $gj$ and connect $gi$ to all fanout gates of $gj$
      recalculate CSPF $G(gi)$ for all $i$
      break
    endif
    $NewG = G(gi) \cap G(gj)$ /* calculate CSPF for a new gate */
    if $NewG \neq \emptyset$ /* case3 */
      if $NewG \supseteqq F(gi)$ and $Case3\_check$ in $gi$ wrt $gj$
        disconnect all connection to $gj$ and connect $gi$ to all fanout gates of $gj$
        recalculate CSPF $G(gi)$ for all $i$
        break
      endif
      if $NewG \supseteqq F(gj)$ and $Case3\_check$ in $gj$ wrt $gi$
        disconnect all connection to $gi$ and connect $gj$ to all fanout gates of $gi$
        recalculate CSPF $G(gi)$ for all $i$
        break
      endif
      $New = \{g | g \in \{gi\}$, $g$ is connectable to $NewG\}$
      if $NewG \supset F(g)$ in $New$ and $Case3\_check$ in $g$ wrt $gi$ and $gj$
        disconnect all connection to $gi$, $gj$ and connect $g$ to all fanout
        gates of $gi$, $gj$
        recalculate CSPF $G(gi)$ for all $i$
        break
      endif
    endif
  endforeach
endwhile
end
```

Fig. 6 Extended gate merge algorithm

**Example.** Fig.7.a shows an SG and Fig.7.b shows a part of the corresponding SI circuit. Since CSPFs of $g2$ and $g3$ are neither $G(g2) \subseteqq G(g3)$ nor $G(g2) \supset G(g3)$ and the conjunction of them is not empty ($G(g2) \cap G(g3) =$ 11*100000000000000), this is in Case3 (condition (3) is satisfied).

At first, we check whether the substitution of $g3$ by $g2$ is possible or not. For condition (1), we must check whether $g2$ has the same value with respect to $g3$ in all states of $\mathsf{ER}(aout+)$ (these are $s1(10011)$ and $s2(10001)$ in Fig.7.a and b). For state $s2$, both $G(g2)$ and $G(g3)$ have the same value. Even though $G(g2)$ is don't care in state $s1$, the logic value itself is the same (it is simply checked because $g2 = \overline{rout} * \overline{ain} * rin$ covers 10011 and 10001). Hence, condition (1) is also satisfied. In addition, since there is no state where $G(g3)$ is a constant and $G(g2)$ is in ER, condition (2) is also satisfied. Finally, since $G(g2)$ has the same constant in all states where $G(g3)$ is constant, condition (4) is also satisfied. Therefore, this substitution does not lead any hazardous behavior (see Fig.7.c).

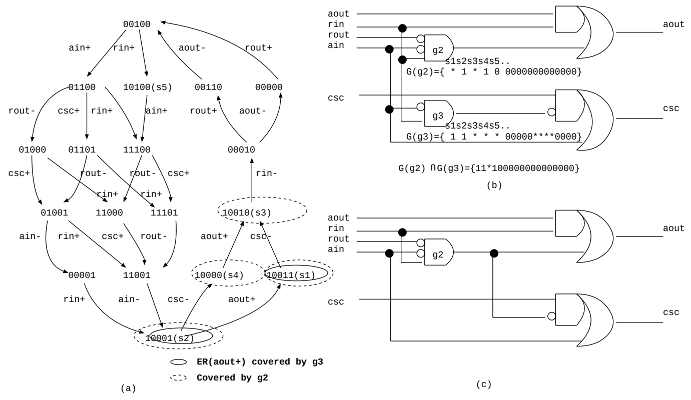


Fig. 7 An example of gate substitutions

Table 1 Results of gC-implementations using gate merge

| 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-forward | 22 | 11 | 29 | 9 | 25 | 2.30 |
| pe-rcv | 69 | 33 | 66 | 32 | 64 | 12.91 |
| ram-read | 39 | 13 | 35 | 12 | 32 | 4.06 |
| alu1_si | 116 | 27 | 52 | 23 | 46 | 23.18 |
| scsi | 141 | 18 | 54 | 17 | 50 | 11.98 |
| total | 463 | 123 | 290 | 108 | 258 | 260.74 |

On the other hand, suppose we check whether the substitution of $g2$ by $g3$ is possible or not. In this case we can see a violation for condition (4) in state s5 (11101) where $G(g2)$ is constant 0 and $G(g3)$ is don't care. Different from $g2$, the logic value of $G(g3)$ in this state is constant 1, hence if we substitute $g2$ by $g3$ we can meet a hazardous behavior in this state. The substitution must be prohibited.

## 5. Experimental Results

In this section, we show the experimental results of our approach applying to the gC-implementations. The gate merge algorithm is implemented using JAVA. It accepts both an SG and the corresponding SI circuit as inputs and produces optimized circuits. The environment of this experiment is Windows98 with Pentium II (300MHz) processor.

Table1 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 circuit, and the five and six ones show the result after optimization respectively. The final column shows the calculation time.

The result shows that we can reduce 12% of the area with respect to the number of nodes (11% wrt the number of literals) on average. This means that on the gC-implementations there are several rooms for the optimization merging common gates or substituting several similar gates.

To validate our transformation we verified the functionality and hazard-freeness using an SI verification tool versify.

There are several considerations through the experiments.

1) Even though the optimization quality compared to `petrify` may be not so big because `petrify` realizes gate-sharing for the common gates during logic decomposition, our approach carries out the optimization globally without introducing logic decomposition. In addition, our approach carries out gate substitution based on CSPFs.

2) But there is no interaction with technology mapping, hence currently our approach may be useful as the post-optimizer of technology mapped circuits. Hence, the application of our approach to the technology mapped circuits will be the future task.

## 6. Conclusion

Because of the hazard-freeness in asynchronous circuits, the resulting circuits after synthesis sometimes suffer from high area overhead. To solve this problem, we proposed an optimization method for asynchronous SI controllers using the transduction method. The experimental results were encouraging in that on average the area reduction by our approach is about 12% in terms of the number of nodes (11% wrt the number of literals). Finally, the algorithm discussed in this paper is implemented using JAVA.

As a future work, we will extend our approach for technology mapped circuits.

### 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 specifications and synthesis of asynchronous controllers. *IEICE Transactions on Information and Systems*, E80-D(3):315–325, March 1997.

[3] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev. Complete state encoding based on the theory of regions. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems.* IEEE Computer Society Press, March 1996.

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

[5] 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.

[6] 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.

[7] 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.