IEICE TRANS. FUNDAMENTALS, VOL.E84–A, NO.8 AUGUST 2001

PAPER

# Automatic Process-Oriented Asynchronous Control Unit Generation from Control Data Flow Graphs

**Euiseok KIM**[†a)], ***Regular Member***, **Jeong-Gun LEE**[†], ***Nonmember***, ***and*** **Dong-Ik LEE**[†b)], ***Regular Member***

**SUMMARY** Although many successful asynchronous control circuit synthesis methods are available, it is still unwieldy to conceive and describe the behaviors of a number of controllers which constitute a control unit of a target system manually. In this paper, an automatic and systematic method to derive an efficient asynchronous control unit from a system specification, a control data flow graph (CDFG), is suggested. In order to acquire an asynchronous control unit of acceptable quality, a new process-oriented method is proposed. In this method, the resulting asynchronous control unit has complete separation of 'execution controllers' and 'execution order controllers' according to the hierarchical decomposition of a given CDFG. This distributive feature leads to a significant improvement in area, performance, implementability and synthesis time for the derived asynchronous control units.

***key words:*** *automatic asynchronous control unit generation, process-oriented, control data flow graphs, signal transition graphs*

## 1. Introduction

In the last decade, many asynchronous logic synthesis methods have been studied and suggested to meet the increased interest in asynchronous circuit design [1]. Although some of them have been implemented as asynchronous logic synthesis tools successfully [2]–[7], it remains difficult to conceive and describe the behaviors for a number of controllers constituting the control unit of a target system manually. For example, when an asynchronous system specification of a target system is first given to a designer, he must describe and implement the asynchronous control unit manually, which may consist of tens of or hundreds of asynchronous controllers. Moreover, an STG, which is a most frequently used and popular specification language for asynchronous controllers, is not user friendly in our opinion. Therefore, we present a way to derive a set of STGs automatically from a given high-level system specification, a CDFG, in order to alleviate the designer's burden in terms of conceiving many STGs for an asynchronous control unit. Derived STGs are synthesized into asynchronous controllers constituting an asynchronous control unit of the target system using an existing asynchronous logic synthesis tool.

Automatic control unit generation has been studied in both synchronous and asynchronous system designs as a part of High-Level Synthesis (HLS). Though many achievements have been made in the automatic generation of associated controllers for synchronous high-level synthesis (SHLS) [8], [9], these cannot be directly applied to asynchronous circuit design due to inherent features such as the absence of a global clock. However, some research has been undertaken on asynchronous high-level synthesis (AHLS). References [10] and [11] introduced a general way of generating an independent control circuit for each hardware component, called a hardware-oriented controller through this paper. Reference [12] presents a compiler from Verilog HDL to an asynchronous controller and a synchronous datapath. In [12], an asynchronous control unit is implemented in a centralized form. In the above two approaches [10]–[12], a set of signal transition graphs (STGs), which are the most commonly used asynchronous control circuit specifications, are derived from a given initial system specification and are synthesized into asynchronous controllers using an asynchronous logic synthesis procedure. Although this work presents complete methodologies for automatic asynchronous control unit generation, the resulting circuits are rather inefficient because their methods cannot guarantee the derivation of small and regular sized STGs satisfying complete state coding, in short CSC, property. Therefore, their methods suffer from area overhead, performance degradation, bad implementability and long synthesis times. For example, in [12], it takes 9.5 CPU hours on an UltraSparc machine to synthesize the control unit of a small sized filter. Moreover, the description in [10], [11] is a simple DFG* having no control constructs such as 'if' and 'while' and thus their description power is very limited. Another method of synthesis, a macromodule based method [13], may cause area overhead due to its inherent redundancy [14].

In this paper, a method for automatic control unit generation from a CDFG is proposed. In order to acquire an asynchronous control unit with competent

Manuscript received August 16, 2000.
Manuscript revised January 10, 2001.

[†]The authors are with the Department of Information and Communications, K-JIST, 1, Oryong-Dong, Puk-Gu, Kwangju, 500-712, Korea.
a) E-mail: uskim@geguri.kjist.ac.kr
b) E-mail: dilee@kjist.ac.kr

*The DFG corresponds to a DFG-unit defined in Def. 1 of this paper.

quality, a new process-oriented method is proposed. A control unit generated by the proposed method has the following five noticeable features;

- to be separated into 'execution controllers' and 'execution order controllers' completely according to hierarchical decomposition of a given CDFG,
- to be uniformly distributed control circuits,
- to guarantee maximal concurrency between processes,
- to be generated from a CDFG through a systematic way,
- to be efficient in the aspects of area, performance, implementability and synthesis time.

Specially, the first feature, complete division of 'execution controllers' and 'execution order controllers' makes other features possible and enables our process-oriented asynchronous control unit generation method to avoid problems which previous approaches experienced.

This paper is organized as follows; Sect. 2 presents preliminaries which are necessary to understand this paper. Section 3 presents and compares several approaches to automatic control unit generation including a process-oriented control unit generation approach. In Sect. 4, we explain how to build a set of communicating controllers from a CDFG using the process-oriented control unit generation method in detail. Section 5 presents timing conditions that a control unit generated by a suggested method should satisfy for the correct operation. In Sect. 6, experimental results to show the effectiveness and correctness of the suggested method are given. Section 7 presents conclusion and future work.

## 2. Preliminaries

### 2.1 Control Data Flow Graphs

We assume that the initial behavioral description of a target system is given in the form of a control data flow graph, in short a CDFG, such as Fig. 1(a) in this paper. Although a CDFG is a bit primitive compared to existing hardware description languages, it can represent the behavior of a target system in a natural and explicit way. Prior to defining a CDFG, two major constituent components, a DFG-unit and a CDFG-unit, are defined. Then we define a CDFG.

**Definition 1:** A DFG-unit is a triple $\Omega = (\mathsf{N}, \mathsf{E}, \mathsf{O})$, where N is a set of operation nodes, E is a set of edges between operation nodes, O is a set of operations defined for each operation node.

**Definition 2:** A CDFG-unit is a 4-tuple $\Gamma = (\mathsf{X}, \mathsf{Y}, \mathsf{Z}, \mathsf{E})$, where X is a control node, Y is a conditional node, Z is the set of nodes in $\Gamma$ except for X and Y, where Z is said a child block of the $\Gamma$'s control node. E is a set of edges between nodes.

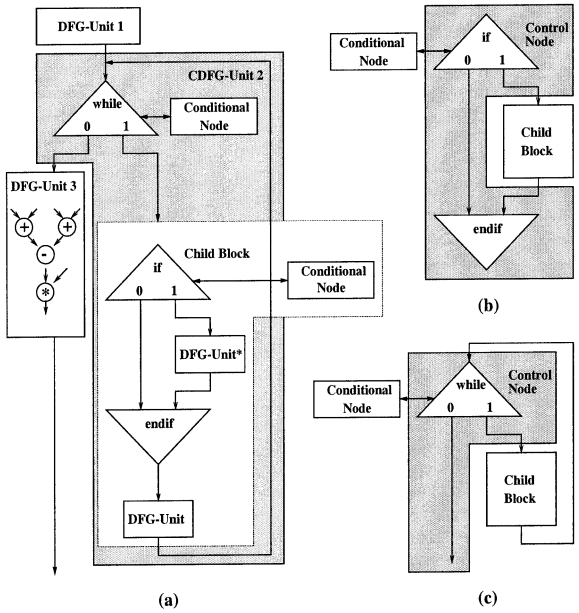



**Fig. 1** (a) A CDFG, (b) IF-node, (c) WHILE-node.

An operation node in a DFG-unit is mapped into a concept of process because an operation consists of 'read operands,' 'execute' and 'write the execution result.' Therefore, we use an operation node and a process in the same meaning in this paper. In a CDFG-unit, the control node is responsible for the control of its child block according to its predefined function. Figures 1(b) and (c) show IF-node and WHILE-node. A conditional node is a DFG-unit representing an execution condition for the control node. A child block corresponds to an execution block under the control of the control node and consists of a set of DFG-units and CDFG-units.

**Definition 3:** A CDFG is a pair $\Sigma = (\mathsf{U}, \mathsf{E})$, where $\mathsf{U}=\Omega \cup \Gamma$ is the set of units and E is the set of edges between units, where a unit is trivially either a DFG-unit or a CDFG-unit. Each unit has at most one predecessor and one successor. Moreover, each unit has at least one predecessor and/or one successor except the case that CDFG consists of only one DFG-unit or one CDFG-unit.

A CDFG is a sequential composition of DFG-units and CDFG-units. Note here that a child block in a CDFG-unit is another CDFG as a matter of fact and thus a CDFG can describe the behavior of a target system hierarchically. In order to implement the CDFG into hardware, details about hardware implementation should be associated to all the DFG-units constituting the initial CDFG as shown in Fig. 2 through scheduling, resource allocation and resource binding procedures. We call a DFG-unit with hardware information a Scheduled, Allocated, Bound-DFG unit, in short an SAB-DFG-unit.

**Definition 4:** An SAB-DFG-unit is a triple $\Omega' = (\Omega, \mathsf{E}', \mathsf{H}')$. Where $\Omega = (\mathsf{N}, \mathsf{E}, \mathsf{O})$ is an initial DFG-unit, $\mathsf{E}'$ is the set of edges representing execution order be-

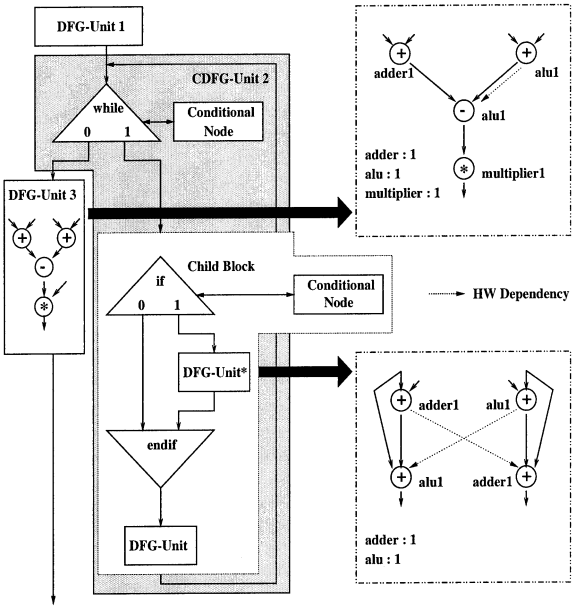



**Fig. 2** A Scheduled, Allocated, Bound-CDFG(SAB-CDFG).

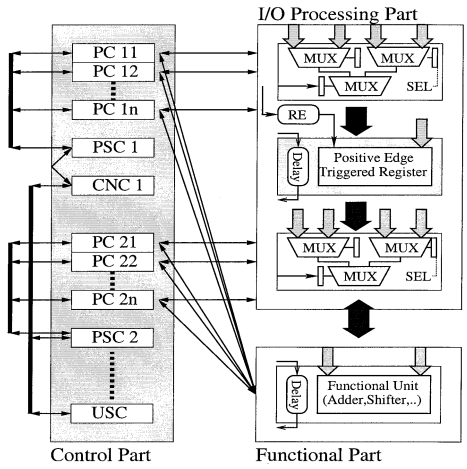



**Fig. 3** A target architecture for automatic asynchronous control unit generation and its process-oriented controllers.

tween the operation nodes sharing the same hardware and $\mathsf{H}'$ is the multiset of hardwares associated with each operation node or each edge in $\Omega$. A CDFG, whose all the DFG-units are SAB-DFG-units, is defined as an SAB-CDFG.

For scheduling, resource allocation and resource binding procedures, there have been little work related to them in asynchronous high-level synthesis to our knowledge. However, there have been a number of good and efficient methods or tools performing them in synchronous high-level synthesis although their exact solutions require intractable computation time. Contrary to automatic synchronous control unit generation, synchronous methods performing them can be used in an asynchronous high-level synthesis. Therefore, we can derive an SAB-CDFG from a CDFG easily through existing synchronous scheduling, resource allocation and resource binding procedures and thus we assume that scheduling, resource allocation and resource binding are performed prior to generating an asynchronous control unit of a target system automatically. That is, we assume that an SAB-CDFG is given as an input specification for automatic control unit generation of a target system. Since an SAB-CDFG is a default input specification, we omit the prefix 'SAB-' for the sake of simplicity in the rest of the paper.

### 2.2 An Architectural Model

The target architecture we consider in this paper is shown in Fig. 3 and is composed of input/output (I/O) processing part, a functional part and a control part.

The I/O processing part consists of input muxes, output muxes and positive edge triggered registers. Furthermore, mux selectors (SEL) which are necessary for choosing an input among mux inputs, register enabling signal generator (RE) for register writing, and delay elements for registers are required. Muxes, which are interconnected to I/O ports of registers, transport data between functional units and registers.

Data processing modules such as adders, ALUs, multipliers and shifters etc. comprise the functional part. In the target architecture, 4-phase handshake protocol and bundled data method for control and data path protocols are assumed respectively for better performance and smaller area. Therefore, each functional unit is implemented in single-rail as synchronous one. Note here that basically bundled data assumption is not essential, but is introduced to reduce the area overhead of functional units.

A control part based on process-oriented control unit design style will be explained in the following sections in detail.

## 3. Approaches to Asynchronous Control Unit Generation

### 3.1 Previous Approaches

The first previous approach is a traditional synchronous control unit generation method. The synchronous control unit is described and synthesized through a finite state machine (FSM), and generates proper control signals at the specified state transition according to a global clock [8], [9]. However, the absence of a global clock in an asynchronous system makes it difficult to apply this method directly. Moreover, the clock based state transitions severely restrict the autonomous and parallel behavior of asynchronous systems.

There are two approaches to asynchronous control unit generation; a centralized control unit generation approach [12], shown in Fig. 4(a) and a hardware-

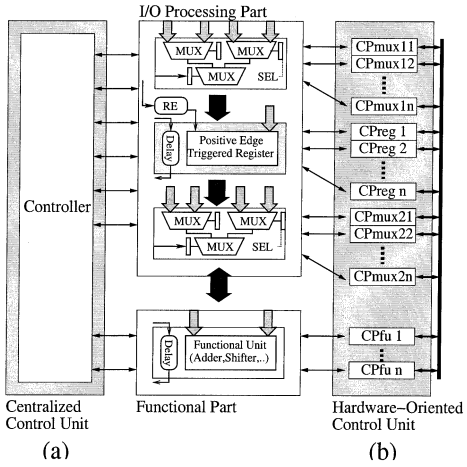



**Fig. 4** (a) A centralized control unit, (b) A hardware-oriented control unit.

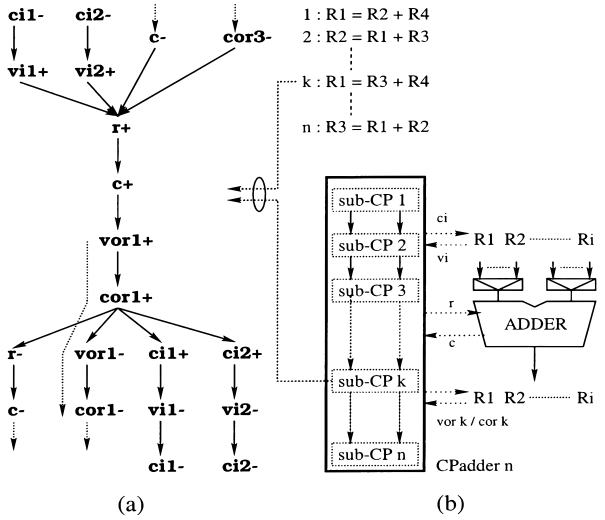



**Fig. 5** (a) A partial STG for a CP, (b) A CP for an adder which $n$ addition operations are allocated to.

oriented control unit generation approach [10], [11], shown in Fig. 4(b). [12] presents a compiler from Verilog to an asynchronous control unit and a synchronous datapath. The control unit is implemented in the centralized form and generates all the signals necessary for system operation. [10], [11] present a hardware-oriented asynchronous control unit generation method. In this method, a set of communicating small controllers, which are called control processes (CPs) and constitute the asynchronous control unit, are derived. A CP is generated for each self-timed hardware block. Since CPs are controllers constituting a hardware-oriented control unit, we call them hardware-oriented controllers in this paper. The CP generates all the control signals necessary for operation of the corresponding hardware block, such as 'operand fetch,' 'execution' and 'writing the execution result.' Moreover, the CP includes information about the execution order of operations allocated to the corresponding hardware block. In the case of a synchronous control unit, execution order is represented in the form of a state in a FSM. That is, state transitions in a FSM describe the execution order of the synchronous control unit. However, an STG represents execution order through sequential or concurrent connections of the control signals required in each state, instead of explicit representation of states. Figure 5(a) shows a partial STG for a CP, which is responsible for performing an operation allocated to an adder. The STG for the CP is built by the sequential composition of the partial STGs as shown in Fig. 5(b). For example, we assume that n additions are allocated to an adder, and then the corresponding STG for the adder is generated through n time compositions of the partial STG in Fig. 5(a) as shown in Fig. 5(b). Therefore, if more work is allocated to a hardware block under the resource constraints, the corresponding STG becomes larger linearly. Although STGs for CPs differ from each other according to their corresponding hardware blocks, their structures and expansions are similar to the STG in Fig. 5. Figure 4(b) shows a hardware-oriented control unit consisting of CPs for functional units, registers and muxes. Although the above two approaches [10]–[12] present complete methodologies for automatic asynchronous control unit generation, the resulting circuits are rather inefficient because special features in STG based logic synthesis, such as state space explosion and complete state coding (CSC) violation problems were not considered.

A state space explosion problem means that the state space of a model is too large to analyze within an acceptable time. The main cause of the state space explosion problem for STGs is their size increase. Although the relation between the size of an STG and the number of states has not been proved theoretically, the number of states of an STG may increase exponentially with size of the STG in many cases. Figure 6 shows the number of states and the synthesis time according to the size of randomly selected STGs from an SIS benchmark set [7]. For Fig. 6, we use an asynchronous logic synthesis tool 'Petrify' [4], in order to obtain the number of states, the synthesis time and the number of literals. SIS benchmarks and 'Petrify' are the most frequently used benchmarks and the most commonly used asynchronous logic synthesis tool respectively. As Fig. 6 shows, the number of states increases exponentially, and hence the synthesis time also increases very sharply according to the increasing size of STGs. For [12], with increasing input system specification size, the STG for a controller becomes larger and more complex because only one centralized controller should control the whole of the system. For [10], [11], a large input system specification causes the sizes of the STGs for CPs to be large, as in the centralized control unit generation method, under the resource constraints, as shown

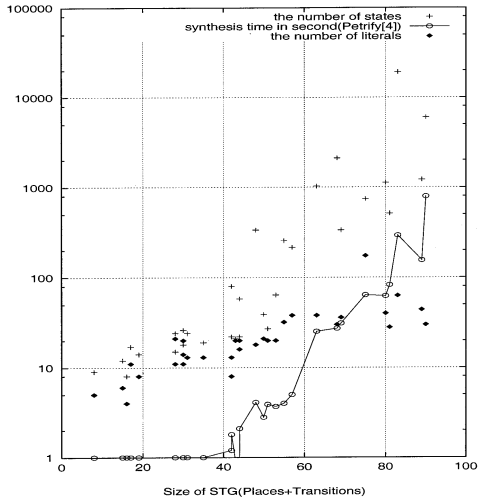



**Fig. 6** The number of states, synthesis time (sec.) and the number of literals of SIS benchmark STGs according to increase of STG size.

in Fig. 5(b). Therefore, their methods eventually suffer from large synthesis times.

If an STG has two states that have the same binary code but a distinct set of enabled non-input signals, then it has CSC violations [15]. CSC violations requiring internal signal insertion for solving them cause a rapid increase in circuit area, delay and synthesis time due to inserted internal signals. The formal definition of CSC property and several problems which CSC violations cause will be explained in detail in Appendix of this paper. Centralized [12] and hardware-oriented [10], [11] asynchronous control unit generation methods do not consider CSC property and assume that they depend on CSC violation solving tools completely. The STG derivation in [10]–[12] is based on a straightforward 4-phase handshake expansion of control signals as shown in Fig. 5(a). Therefore, if careful consideration is not given to CSC property, a derived STG may have a number of CSC violations. For example, the simple 4-phase handshake expansion of control signals such as $vo_{r1}+ \rightarrow co_{r1}+ \rightarrow vo_{r1}- \rightarrow co_{r1}-$ in Fig. 5(a) causes a CSC violation because different sets of output signals, '$vo_{r1}+$' and '$r-$, $ci_1+$ and $ci_2+$,' can be enabled at the same binary codes. Moreover, although Petrify, which is the state of art in solving CSC violations, is used, many internal signals are inserted, which results in area and speed inefficient circuits consuming much time. Moreover, sometimes CSC violations cannot be solved by the tool. Therefore, these methods [10]–[12] suffer from CSC violations eventually.

### 3.2 A Process-Oriented Approach

The process-oriented control unit generation method shown in Fig. 3 is a new way to derive a decentralized asynchronous control unit according to hierarchical decomposition of a CDFG. A process-oriented control unit shown in Fig. 3 is composed of process controllers (PCs), process sequencing controllers (PSCs), control node controllers (CNCs) and unit sequencing controllers (USCs) that communicate with each other. The first two controllers, PCs and PSCs, work as controllers for a DFG-unit and the rest, CNCs and USCs, take responsibility for controlling higher level than DFG-units in a CDFG.

A PC, which is built for a process in a DFG-unit, generates all the control signals necessary for executing one process in the DFG-unit. Since each process is regular in sense of behavior, that is each process consists of 'read operand,' 'execute' and 'write the execution result,' sizes of PCs are also regular and small. A PSC corresponds to a DFG-unit and coordinates execution order among PCs based on dependencies in the DFG-unit. Therefore, a PSC is considered as a centralized one basically, while PCs are distributed for a DFG-unit. A CNC corresponds to a control node in a CDFG-unit and thus there are two kinds of CNCs, WHILE-CNC and IF-CNC. Each CNC handles a child block under its control according to the function of the corresponding control node in a CDFG-unit. A USC coordinates execution order among CNCs and PSCs as a PSC does among PCs. As previously mentioned, the important distinctive feature of a process-oriented control unit is complete separation of 'execution controllers' and 'execution order controllers' according to hierarchical decomposition of a given CDFG. Actually, that means that a process-oriented asynchronous control unit consists of PCs, PSCs, CNCs and USCs explained above. That is, PCs correspond to 'execution controllers' and the others correspond to 'execution order controllers.' Figure 7(a) presents hierarchical structure of process-oriented control unit for Fig. 2 and Fig. 7(b) shows general hierarchy of process-oriented controllers.

STGs for 4 kinds of process-oriented controllers which are derived from a hierarchical decomposition of a CDFG are always small and Sect. 4 will show that fact explicitly. Therefore, the process-oriented method do not experience the state space explosion inherently. Moreover STGs for 4 kinds of process-oriented controllers satisfy CSC property inherently through careful signal coding in STG derivation procedure or algorithm. In the following section, for each STG, it will be proved[†] that each STG satisfies CSC property inherently. The fact that no state space explosion and no CSC violation do not occur in our process-oriented method is a very important advantage of our method over previous approaches. Therefore, you should understand how those features can be achieved or guaranteed in the following section. Note here that a hardware-oriented

[†]For CSC property, a proof in Prop.1 shows that process-oriented controllers have no CSC violations through careful interleavings of signal transitions inherently.

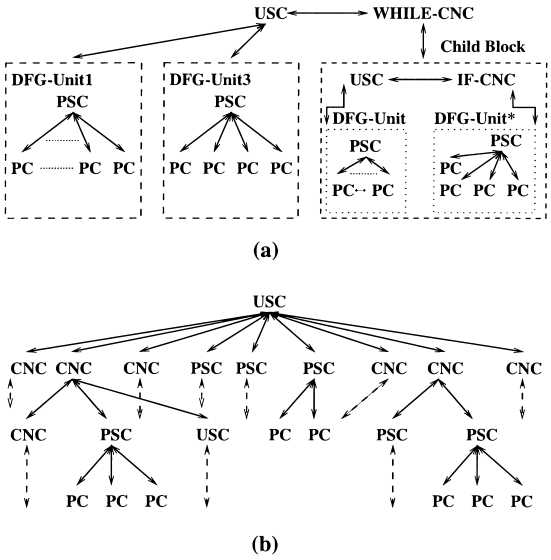



**Fig. 7** (a) Hierarchical structure of process-oriented control unit for Fig. 2, (b) General hierarchy of process-oriented controllers.

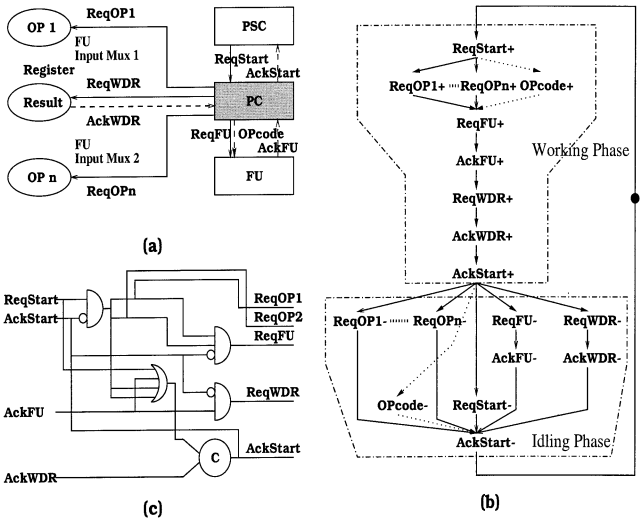



**Fig. 8** (a)Signal exchanges in a PC, (b) An STG for a PC, (c) Gate-level implementation of the STG given in (b).

method and our process-oriented method use a different decomposition criterion. That is, the former derives a control unit decomposed according to hardware component blocks but the latter derives a control unit decomposed according to processes in a DFG-unit and hierarchical interconnection of DFG-units and CDFG-units. Therefore, degree of decomposition in the hardware-oriented method varies according to hardware allocation but degree of decomposition in the process-oriented method is always uniform. Moreover, the hardware-oriented method is restricted to DFG specifications but our process-oriented method can handle CDFGs.

## 4. Generation of Process-Oriented Controllers

In this section, we explain how to derive a distributed control unit based on a process-oriented method in detail. The first, STGs for controllers are derived and then they are synthesized into asynchronous control circuits.

### 4.1 Generating Process Controllers

A process controller, in short PC, is a module to generate proper control signals necessary for performing a process corresponding to an operation node in a DFG-unit. In order to perform a process, a PC exchanges several control signals with a PSC, a functional unit, muxes and registers as shown in Fig. 8(a). The general behavior of a PC is described as follows;

step 1 A PC is activated by receiving ReqStart+ from the associated PSC.
step 2 The PC sends request signals necessary for operand fetch such as $\mathrm{ReqOP}_1+$ and $\mathrm{ReqOP}_n+$. If necessary, OPcode+ is generated as an OP code signal.
step 3 The PC activates the bound functional unit by sending a ReqFU+.

Since all signal exchanges are based on a 4-phase bundled handshake protocol, AckFU+ arrives at the PC after some fixed delay. Timing constraints for correct operation is discussed in Sect. 5 in detail.

step 4 The PC generates ReqWDR+ and sends it to the destination register in order to store the execution result.

Similar to AckFU+, AckWDR+ arrives at the PC after some fixed delay.

step 5 After whole procedures of a process have been completed, the PC sends AckStart+ to the PSC and enters an idling phase immediately.

Behavior of a PC is described in a signal transition graph (STG) that has been widely accepted as a description language of asynchronous control circuits after introduced in [16]. Figure 8(b) shows an STG corresponding to PC's general behavior. STGs for PCs may be different according to operations of the corresponding process. For example, in the case of an assignment operation, rising/falling transitions of $\mathrm{ReqOP}_2 \cdots \mathrm{ReqOP}_n$, OPcode, ReqFU and AckFU are not necessary. Note that the size of a PC is always similar to the size of the STG in Fig. 8(b) because a PC is a simple controller to perform an operation node in a DFG-unit. The circuit in Fig. 8(c) is one of PC example circuits derived from STGs by using asynchronous control circuit synthesis tool, Petrify [4]. In general, an STG should satisfy the following four properties to guarantee a speed-independent operation of the generated circuit; boundedness, output semi-modularity, consistency and

CSC [15], [17]. For the lack of space, a formal definition of each property is omitted here. Please refer to [15] and [17] for the definition. Detailed explanation about CSC property is given in Appendix.

**Proposition 1:** An STG for a PC always satisfies boundedness, consistency, output semi-modularity and CSC properties.
[proof] All the proofs are almost trivial.
boundedness: An STG for a PC is a strongly connected marked graph, in short MG, from the construction and thus the corresponding STG is structurally bounded [18].
consistency: The STG has exactly one rising transition and one falling transition for each signal. Moreover, rising and falling transitions for a signal always occur alternatively. Therefore, consistency is satisfied.
output semi-modularity: An enabled transition in an MG can not be disabled by firing of other transitions. Therefore, the STG satisfies output semi-modularity inherently.
CSC: The STG has a working phase and an idling phase explicitly. Namely, all the rising transitions occur in a working phase and all the falling transitions occur in an idling phase. Since a working phase ends in AckStart+ and an idling phase ends in AckStart-, all the states generated in a working phase have different binary states from ones generated in an idling phase. Moreover, all the states in the same phase have different binary states from each other. Therefore, all the binary states generated from the STG are distinct. Therefore the STG satisfies CSC property.

According to Prop.1, an STG for a PC can be synthesized into a speed-independent circuit without any modifications.

### 4.2 Generating Process Sequencing Controllers

A process sequencing controller, in short PSC, is a circuit activating a series of PCs in a proper order based on the data or resource dependencies among processes or operation nodes in a DFG-unit. In the first step of automatic PSC generation, we transform a DFG-unit into a Petri net, in short PN. Since all the nodes in a DFG-unit has been already scheduled, the corresponding PN can be easily obtained.

**Definition 5:** For any two operation nodes, $n_i$ and $n_j$, in a DFG-unit, if there exists an edge $e_k$ or $e'_k$ from $n_i$ to $n_j$, $n_i$ precedes $n_j$, denoted by $(n_i P n_j)$, where $e_k$ and $e'_k$ are edges to denote data dependency relation and resource dependency relation respectively. If $(n_i P n_j)$ and there does not $n_k$ such that $(n_i P n_k)$ and $(n_k P n_j)$, then $n_i$ directly precedes $n_j$, denoted by $(n_i DP n_j)$.

Definition 5 defines the precedence relation between two operation nodes connected by edges $e_k$ or $e'_k$ in the aspects of data or resource dependencies. Based on Def. 5, the following is an algorithm to construct a PN from a DFG-unit.

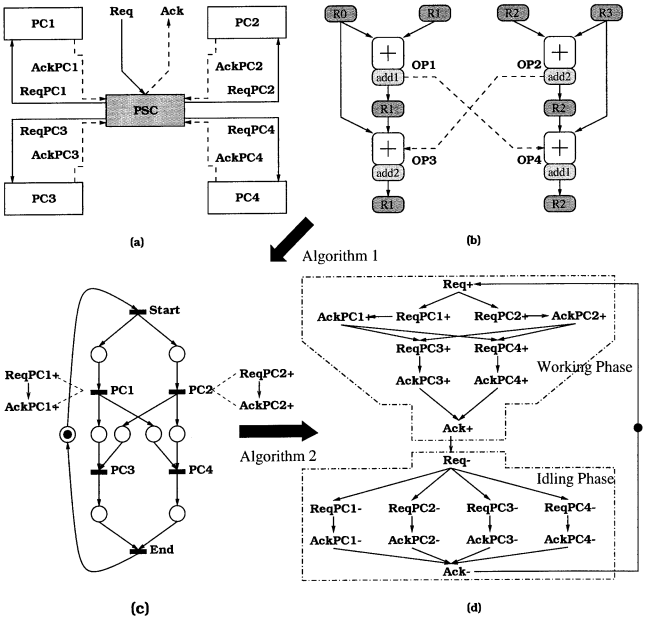



**Fig. 9** (a) Signal exchanges in a PSC, (b) An SAR-DFG-unit, (c) A PN derived from an SAR-DFG-unit in (b), (d) An STG for a PSC.

**Algorithm 1: Derivation of a PN from a DFG-unit**
Note that any DFG-unit does not have choice inherently and thus the derived net is an MG. Therefore, "make an arc from t to t′" in this algorithm means that t and t′ are connected via a place p, i.e., $t \rightarrow p \rightarrow t'$.
step 1 Generate two transitions labeled Start and End.
step 2 For each operation node $op_i$, make a corresponding transition and label it as $PC_i, i = 1, 2, \ldots$.
step 3 For each transition $PC_i$ corresponding to an operation node $op_i$ which has no operation node $op_j$ such that $(op_j P op_i)$, make an arc from Start transition to $PC_i$.
step 4 For each transition $PC_i$ corresponding to an operation node $op_i$ which has no operation node $op_j$ such that $(op_i P op_j)$, make an arc to End transition.
step 5 Make an arc from $PC_i$ to $PC_j$ corresponding to operation nodes, $op_i$ and $op_j$ such that $(op_i DP op_j)$.
step 6 Make an arc from End transition to Start transition and put a token at a place on the arc.

Moreover the PN derived from a DFG-unit can be automatically transformed into an STG in a straight forward way. The following algorithm shows derivation of an STG for a PSC from a PN using 4-phase handshaking protocol. Figures 9(b), (c) and (d) show how to derive a PN and an STG from a DFG-unit through Algo.1, 2 and Fig. 9(a) shows signal exchanges between PSC and associated PCs.

**Algorithm 2: Derivation of an STG from a PN Using 4-phase Handshaking protocol**

step 1 Transform the Start transition and the End transition to Req+ and Ack+⟶Req−, respectively.

step 2 Divide each transition corresponding to $PC_i$ into two rising signals $ReqPC_i+$ and $AckPC_i+$ and add an arc as $ReqPC_i+ \longrightarrow AckPC_i+$.

step 3 For each signal generated in steps 1 and 2, make an arc from $AckPC_i+$ to $ReqPC_j+$ if there is an arc from $PC_i$ to $PC_j$ in the PN.

step 4 For each rising transition $ReqPC_i+$ corresponding to $PC_i$ which is a postnode of Start transition, make an arc from Req+ to $ReqPC_i+$. And then for each rising transition $AckPC_j+$ corresponding to $PC_j$ which is a prenode of End transition, make an arc from $AckPC_j$ to Ack+.

step 5 Make falling signals $ReqPC_i-$'s, $AckPC_i-$'s and Ack− and add arcs as $ReqPC_i- \longrightarrow AckPC_i-$.

step 6 Make arcs from Req- to all the $ReqPC_i-$'s and from all the $AckPC_i-$'s to Ack−.

step 7 Add an arc from Ack- to Req+ and put a token on the arc.

It can be proved in a similar way to Prop.1 that STGs obtained through Algo.2 always satisfy the four properties for speed-independent circuit implementation. Therefore, an STG for a PSC can be also synthesized into a speed-independent circuit without any modifications. Moreover, a PSC, which is derived through Algo. 1 and 2, guarantees maximal concurrency between processes in a DFG-unit.

There are two points that are worth noting in construction of a PSC, the associated PCs and their interactions: Consider a PSC, say $PSC_i$, and all associated PCs, say $PC\_PSC_i$. $PSC_i$ activates every PC in $PC\_PSC_i$ in the order of associated dependencies before entering the idling phase. Then each PC in $PC\_PSC_i$ executes the assigned process. After completion of the process, the PC sends out an acknowledge signal. Then the PC immediately goes into its idling phase, i.e., pulling down request and acknowledgement signals, by its own control rather than by $PSC_i$'s idling phase. In consequence, we can enjoy better performance by hiding idling phases of all PCs. For example, in the STG shown in Fig. 9(d), the PSC issues $ReqPC_i+$, i=1, 2, to activate $PC_1$ and $PC_2$. $ReqPC_i+$ corresponds to ReqStart+ in the STG for a PC as shown in Fig. 8(b). After the completion of the assigned process $PC_i$ sends AckStart+ which is identical to $AckPC_i+$ in the PSC. Then $PC_i$ immediately starts the idling phase without any interaction with the PSC by issuing $ReqOP_1-, \cdots ReqOP_n-$, ReqFU- and so on. At the same time the PSC sends another $ReqPC_3+$ and $ReqPC_4+$ without issuing $ReqPC_i-$, i=1, 2.

In addition, if decomposition of a PSC is required for such reasons as synthesis time and implementability, for example if several small sized PSCs are preferable rather than one large PSC, a PSC can be decomposed easily into small sub-PSCs which communicate with each other and control allocated PCs in the structure shown in Fig. 10. A decomposed PSC behaves in the exactly same manner as an original PSC and this decomposition guarantees further distribution of control circuits in small and uniform size.

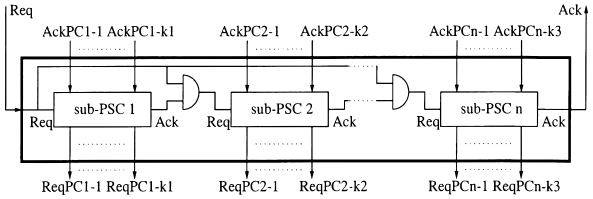



**Fig. 10** A structure of a decomposed PSC.

## 4.3 Building Control Node Controllers

In a CDFG, control nodes coordinate executions of child blocks. In this paper, we assume that there are two kinds of control-nodes according to their function; IF-node and WHILE-node. IF-node performs the child block under its control if the given condition is true. Similarly, WHILE-node executes the child block under its control while the given condition is true.

A control node controller, in short a CNC, corresponds to a control node in a CDFG-unit. A control node operates according to the given condition and thus a CNC should check whether the given condition is true or not through executing a conditional node when an input signal Req+ is activated. Then, as a result of executing the conditional node, the CNC activates the child block according to the value of Flag which indicates an execution result of the conditional node. Figure 11(a) shows a block diagram which shows signal exchanges among CNC, the associated conditional node and the child block. We propose STGs for IF-CNC and WHILE-CNC as shown in Figs. 11(b) and (c), respectively. STGs for IF-CNC and WHILE-CNC satisfy four properties for speed-independent circuit synthesis by construction. Therefore, they can be synthesized into speed-independent circuits without any modifications. Although we define only two CNC controllers, IF-CNC and WHILE-CNC, for the simplicity, the set of CNC controllers can be expanded easily at need.

## 4.4 Building Unit Sequencing Controllers

A unit sequencing controller, in short a USC, is to coordinate the execution order among CNCs and PSCs in order to perform a CDFG as a PSC does for a DFG-unit. According to the definition of a CDFG, DFG-units and CDFG-units constituting a CDFG are performed sequentially. Therefore, executions of CNCs and PSCs that are handled by a USC are linearly ordered. The following algorithm shows the procedure which generates a

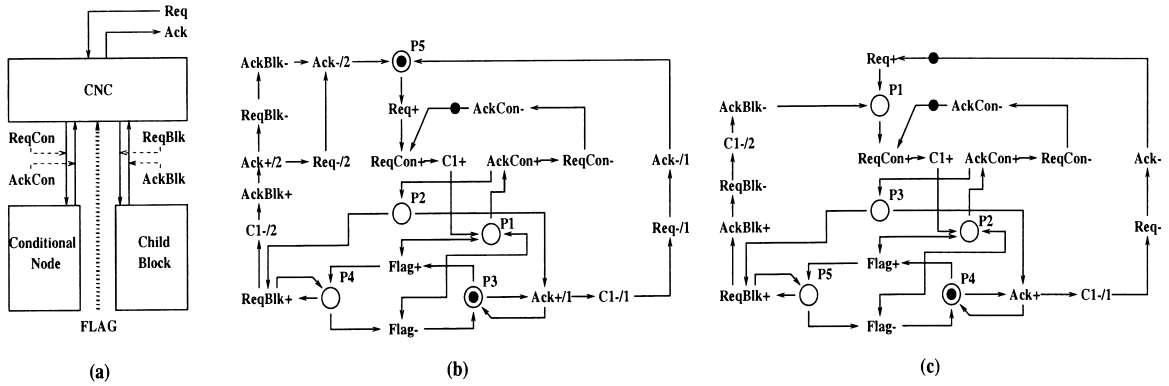



**Fig. 11** (a) Signal exchanges in CNC, (b) An STG for IF-CNC, (c) An STG for WHILE-CNC.

PN representing the execution order among CNCs and PSCs from a CDFG. Note that "make an arc" in the algorithm also means that two transitions are connected via a place.

**Algorithm 3: Derivation of a Sequential PN from a CDFG**

step 1 Generate two transitions Start and End.
step 2 For every DFG-unit or CDFG-unit, make a corresponding transition and label it as $\text{Block}_i$, i=1, 2, ....
step 3 Make an arc from Start transition to a transition corresponding to a DFG-unit or a CDFG-unit having no predecessor.
step 4 Make an arc to End transition from a transition corresponding to a DFG-unit or a CDFG-unit having no successor.
step 5 For two adjacent units in a CDFG, make an arc between corresponding two transitions according to the given dependency. Here a unit means a DFG-unit or a CDFG-unit.
step 6 Make an arc from End transition to a Start transition and put a token at a place on the arc.

A sequential PN obtained through Algo.3 can be transformed into an STG of the target USC in the same way as Algo.2. It can be proved in a similar way to Prop.1 that the resulting STG satisfy the four properties for speed-independent circuit implementation. Therefore, the STG can be synthesized into a speed-independent circuit without any modifications like PSC. Figure 12 shows a series of steps to derive an STG for a USC from a CDFG. If the size of system specification becomes large, the size of a USC may be large. In this case, the USC can be also decomposed into small sub-USCs in the same manner as PSC's decomposition.

## 5. Timing Constraints for Correct Operation of Process-Oriented Asynchronous Control Units

Timing constraints are necessary for correct control of

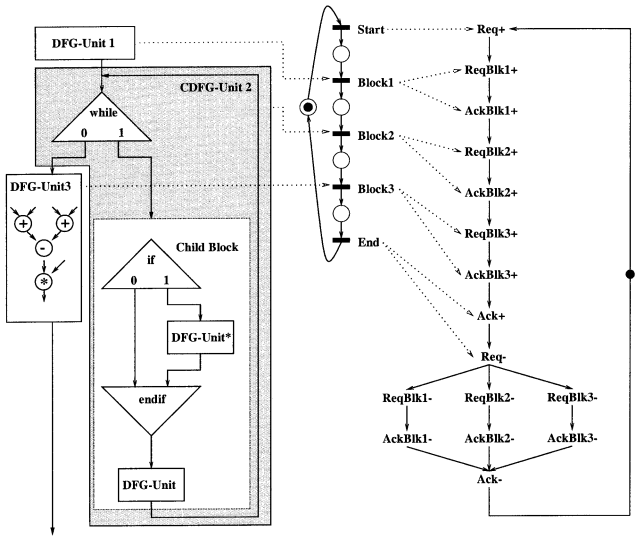



**Fig. 12** Derivation of an STG for a USC from a CDFG.

whole system by a proposed control unit. Those constraints are as follows;

1. The size of delay associated to a functional unit (FU) should be larger than the sum of maximum operand fetch delay, the FU's worst case delay and worst case delay of destination register's input muxes.
2. The size of delay associated to a register should be larger than delay for register writing.
3. For two consecutive processes, $p_i$ and $p_j$, using the same hardware, if $p_i$ is executed prior to $p_j$, the idling phase of $p_i$ should not overlap with the working phase of $p_j$.

Since we assume that bundled data method is used for small area, necessities of the first and the second constraints are trivial and it is easy to satisfy 1 and 2. I/O ports of a FU are connected to interconnection logics consisting of a series of muxes. Therefore, maximum operand fetch delay and worst case delay of destination register's input muxes should be considered in

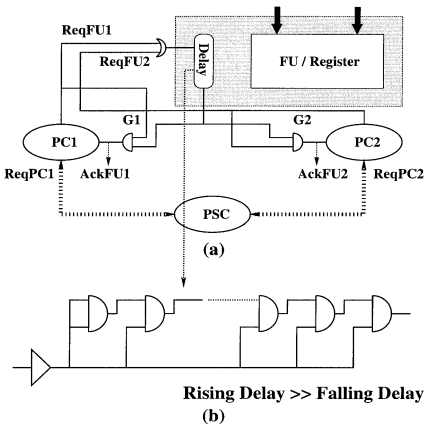



**Fig. 13** (a) Signal exchanges based on 4-phase bundled data method between FU/Register and PC, (b) A delay element with fast high-to-low propagation delay.

constraint 1. The last constraint is necessary for correct implementation of 4-phase handshaking protocol based on bundled data method. Note that signal exchanges between FU/Register and PC are implemented as shown in Fig. 13. We assume that $PC_1$ and $PC_2$ are controllers to access the same FU sequentially as shown in Fig. 13(a). Consider the FU has completed for $ReqFU_1+$. Then delay element gives $AckFU_1+$ to $PC_1$. Since $G_1$ and $G_2$ share output which comes from delay element D, if $PC_2$ sends $ReqFU_2+$ before the output of delay element D becomes low, $PC_2$ receives $AckFU_2+$ immediately. This problem originally stems from the fact that a PSC activates all associated PCs before it enters into the idling phase for better performance. To avoid this situation, we should insert a delay element with proper delay between PSC and $PC_2$. This causes some performance degradation. In order to avoid or reduce this performance degradation due to the third constraint, we make idling phases of PCs as short as possible through concurrent signal fallings and a specialized delay element whose delay time is very small for a falling signal as shown in Fig. 13(b). Consequently, the third constraint can be satisfied easily with only small performance loss.

## 6. Experimental Results and Discussions

In this paper, we suggest a process-oriented control unit generation method. The proposed method has been implemented as a part of an asynchronous high-level synthesis tool. This tool derives a set of STGs for asynchronous controllers based on a process-oriented method and then synthesizes them into speed-independent circuits. In the logic synthesis procedure, Petrify [4] is used. In order to check the effectiveness and correctness of the suggested method, we performed three experiments. In the first two experiments, a hardware-oriented method and a process-oriented method were compared with respect to area, performance, implementability and synthesis time of their resulting circuits. In the last experiment, we applied a process-oriented method to the design of an asynchronous control unit of a '*differential equation solver*.'

**Table 1** General hardware-oriented controller benchmarks.

| name | pl+tr | lits | csc sigs | csc lits | fanin 6,7 | fanin ≥8 | synthesis time |
|---|---|---|---|---|---|---|---|
| $CP_{fu}1$ | 47 | 28 | 3 | 15 | 1 | 0 | 17.7 sec. |
| $CP_{fu}2$ | 70 | 60 | 6 | 23 | 2 | 0 | 204.6 sec. |
| $CP_{fu}3$ | 89 | 75 | 9 | 43 | 3 | 0 | 664.07 sec. |
| $CP_{fu}4$ | 112 | 185 | 12 | 80 | 4 | 2 | 2273.45 sec. |
| $CP_{fu}5$ | 128 | 325 | 14 | 87 | 13 | 5 | 6262.5 sec. |
| $CP_{reg}1$ | 52 | 25 | 4 | 15 | 1 | 0 | 28.3 sec. |
| $CP_{reg}2$ | 81 | 70 | 5 | 28 | 0 | 2 | 213.87 sec. |
| $CP_{reg}3$ | 105 | 201 | 8 | 59 | 2 | 5 | 1397.9 sec. |
| $CP_{reg}4$ | 140 | 187 | 13 | 76 | 1 | 4 | 4502.0 sec. |
| $CP_{reg}5$ | 175 | 354 | 15 | 88 | 6 | 7 | 9140.3 sec. |

'pl' and 'tr' denote the number of places and transitions in STGs.
'lits' is the number of literals of a synthesized controller.
'csc sigs' is the number of internal signals which are inserted to solve CSC violations.
'csc lits' is the number of literals constituting internal signals.
'fanin' is the number of inputs for a gate.

**Table 2** General process-oriented controller benchmarks.

| name | pl+tr | lits | csc sigs | csc lits | fanin 6,7 | fanin ≥8 | synthesis time (sec.) |
|---|---|---|---|---|---|---|---|
| $PC_c$ | 37 | 15 | 0 | 0 | 0 | 0 | 1.25 sec. |
| $PC_a$ | 17 | 5 | 0 | 0 | 0 | 0 | 0.23 sec. |
| $PSC_2$ | 26 | 4 | 0 | 0 | 0 | 0 | 0.51 sec. |
| $PSC_4$ | 46 | 14 | 0 | 0 | 0 | 0 | 2.82 sec. |
| $PSC_8$ | 86 | 30 | 0 | 0 | 0 | 1 | 58.59 sec. |
| $PSC_8^*$ | 92 | 28 | 0 | 0 | 0 | 0 | 5.64 sec. |
| IF − CNC | 41 | 26 | 0 | 0 | 0 | 0 | 1.59 sec. |
| WHILE − CNC | 34 | 27 | 0 | 0 | 0 | 0 | 1.21 sec. |
| $USC_2$ | 25 | 6 | 0 | 0 | 0 | 0 | 0.45 sec. |
| $USC_4$ | 43 | 12 | 0 | 0 | 0 | 0 | 2.14 sec. |
| $USC_8$ | 79 | 24 | 0 | 0 | 0 | 1 | 54.84 sec. |
| $USC_8^*$ | 86 | 24 | 0 | 0 | 0 | 0 | 4.28 sec. |

$*$ denotes decomposed PSC or USC versions.

In the first experiment, we compared basic controllers, which form hardware-oriented and process-oriented control units, as shown in Tables 1 [10] and 2 respectively. Hardware-oriented and process-oriented control units consist of different basic controllers. A hardware-oriented control unit has CP's for functional units, registers and multiplexers as basic controllers. On the other hand, PC's, PSC's, CNC's and USC's are all basic controllers in a process-oriented control-unit. Roughly speaking, for $CP_{fu}$, which is a hardware-oriented controller dedicated to a functional unit 'fu,' whatever the functional unit 'fu' may be, the size and structure of the $CP_{fu}$ are almost same. Likewise, for basic process-oriented controllers, the size and structure of each kind of basic controllers are almost the same. However, in the case of a basic hardware-oriented controller, the size relies on the number of executions of a given functional unit. In the case of a basic process-oriented controller such as a PSC or a USC, the size of the PSC is decided upon by the number of PC's coordinated by the PSC, and the size of the USC is decided upon by the number of CNC's and PSC's coordinated

by the USC. Table 1 shows that when the number of executions increases, the corresponding controller gets more complex and Table 2 shows that when the number of PC's, PSC's and CNC's increases, the coordinating controller, a PSC or a USC, also gets more complex. In Table 1, $\mathsf{CP_{fu}K}$, K=1, 2, $\cdots$, 5, denotes a hardware-oriented controller, which manages a functional unit 'fu' that performs an operation K times, and $\mathsf{CP_{reg}K}$, K=1, 2, $\cdots$, 5, is a hardware-oriented controller for a register 'reg' performing a write operation K times. In Table 2, $\mathsf{PSC_K}$ and $\mathsf{USC_K}$ denote controllers, which coordinate K PCs and K PSCs and CNCs respectively. Hardware-oriented and process-oriented control units mainly consist of basic controllers as shown in Tables 1 and 2. Since a control unit is composed of several basic controllers, the advantages and disadvantages of controllers shown in Tables 1 and 2 directly affect control units. Therefore, the analysis data in Tables 1 and 2 obtained from the first experiment supports our claim on the superiority of process-oriented control units, obtained using our method, over hardware-oriented control units generally.

We explain the correspondence between controllers in Tables 1 and 2. Consider a DFG-unit performing K same operations to which one functional unit is allocated. In this case, a hardware-oriented control unit for the DFG-unit consists of $\mathsf{CP_{fu}K}$, $\mathsf{CP_{reg}K1}$'s and $\mathsf{CP_{mux}K2}$'s, but we consider only $\mathsf{CP_{fu}K}$ for the sake of simplicity. On the other hand, the corresponding process-oriented control unit consists of K $\mathsf{PC_c}$'s and $\mathsf{PSC_K}$. Roughly, for a given DFG-unit, hardware-oriented and process-oriented control units have correspondence of $\mathsf{CP_{fu}K}$ : $\mathsf{K{\cdot}PC_c{+}PSC_K}$. Here, $\mathsf{PSC_K}$ is always a decomposed version if K is large because decomposition is always possible and easily performed, as explained in Sect. 4.2. For the hardware-oriented controllers in Table 1, the number of places, transitions, and literals, and internal signals, high-fanin gates and the synthesis time increase very rapidly, as shown in Table 1, while those of the process-oriented controllers increase slowly, and internal signals and high-fanin gates are not required, as shown in Table 2. Moreover, the sizes of STGs for process-oriented controllers are maintained at a proper size without increasing continuously. Therefore, we know that hardware-oriented control units corresponding to $\mathsf{CP_{fu}K}$ are worse than process-oriented control units corresponding to $\mathsf{K{\cdot}PC_c{+}PSC_K}$. Moreover, with increasing size of given CDFGs, the difference in terms of numbers of literals, area, high-fanin gates and synthesis time becomes larger and larger. The analysis of controllers in Tables 1 and 2 illustrates this explicitly.

As previously mentioned, the number of states for an STG may increase exponentially with respect to the size of the STG in most cases. Moreover, with the increase of one signal in an STG, the size of state space, which is required in asynchronous logic synthesis, grows twice and hence the synthesis time increases very rapidly. For example, consider an STG with $3n$ signals and three STGs with $n$ signals respectively. The size of state space required in asynchronous logic synthesis for the former STG with $3n$ signals is $2^{3n}$. However, the size of state space required in asynchronous logic synthesis for the latter STGs is $3{\cdot}2^n$. This fact implies that the corresponding synthesis time increases exponentially with a linear increase in its size of an STG, and thus the synthesis time of several small STGs is much smaller than the synthesis time of a large STG. Figure 6 shows that fact very well. In Sect. 3, we mentioned that STGs for hardware-oriented controllers have a number of CSC violations. Therefore, the number of inserted internal signals for solving CSC violations and the synthesis time increase rapidly with an increase of K for $\mathsf{CP_{fu}K}$'s or $\mathsf{CP_{reg}K}$'s in Table 1. The above two facts, i.e., the exponential increase of the number of states and CSC violations, have very serious consequences because K and a corresponding STG tend to increase for a large system specification under the hardware resource constraints. For example, in the case of $\mathsf{CP_{fu}5}$, it takes long time, 6262.5 seconds, for synthesis. Moreover, 14 internal signals are inserted for solving CSC violations with additional area overhead. The number of literals necessary for implementing inserted internal signals in Table 1 shows how much area overhead is required in the hardware-oriented method. These internal signals cause performance degradation by introducing additional transitions of internal signals. Although Petrify [4], which is the state of the art in asynchronous logic synthesis and solving CSC violations, was used in this experiment, many internal signals were inserted. Here, it is worth pointing out another problem, 'implementability.' Implementability means how easily designers can implement correct asynchronous control units. As is shown in Table 1, gates with 6, 7 or more fanins are needed for hardware-oriented controllers. This feature is due to the fact that a traditional logic synthesis methodology, as used by Petrify, assumes 2-level sum-of-product implementation for set or reset inputs of a C-element for correct synthesis. Therefore, synthesized circuits of large STGs may have a number of high-fanin gates because of the restricted circuit structure. Contrary to synchronous controllers, it is difficult or impossible to decompose a high-fanin gate into small-fanin gates because of hazards under the current logic synthesis technology. Therefore, if a cell library does not support high-fanin gates, a designer should implement all the hardware-oriented controllers with high-fanin gates in the full-custom style instead of using the cell library. For the process-oriented approach, Table 2 shows analysis data for four kinds of controllers, PCs, PSCs, CNCs and USCs in the process-oriented method. In particular, PSCs and USCs are scalable according to the size of the given DFG-unit or CDFG. Thus we experimented with several PSCs and USCs at various sizes. As

shown in Table 2, the sizes of the STGs for the process-oriented controllers, except PSCs and USCs, are small compared to hardware-oriented controllers, and thus the synthesis takes comparatively little synthesis time. For PSCs and USCs, their sizes also tend to get larger if a system specification is large. However, they can be decomposed into small sub-controllers as explained in Sects. 4.2 and 4.4. Note here that the size of an STG for a decomposed PSC or USC does not refer to a single STG but a summation of all the STGs' sizes for sub-PSCs and sub-USCs. Therefore, process-oriented controllers can be synthesized within a short time because of the small size of the corresponding STGs. In terms of CSC property, all kinds of process-oriented controllers satisfy CSC property inherently. Moreover, all process-oriented controllers can be implemented with only 5 or fewer fanin gates because of the small sizes of the corresponding STGs. Therefore, the process-oriented control unit generation method presents a way of generating an efficient asynchronous control unit without experiencing 'big synthesis time,' 'area overhead,' 'performance degradation' and 'bad implementability.' Note that those advantages of the process-oriented method result from the fact that it can always guarantee the derivation of small sized STGs satisfying the CSC property inherently, as shown in Table 2.

In the second experiment, we applied a hardware-oriented method and a process-oriented method to the well-known high-level synthesis benchmarks, FIR-filters and IIR-filters. The derived control units were implemented at the gate level using a commercial VHDL tool, SYNOPSYS, and 0.6 $\mu$m cell library and then simulated using CADENCE. In this experiment, we compared the number of literals, area, average input/output response time, as denoted by the average I/O RT in the table, and the synthesis time; Table 3 shows the analysis data. In terms of area, the hardware-oriented control units were about 5 times larger than the corresponding process-oriented control units. In our opinion, no CSC violation and good decomposition between 'execution controllers' and 'execution order controllers' without redundancy lead to these good results. In particular, as shown by the analysis data of hardware-oriented controllers in Table 1, internal signals for solving CSC violations cause big area overheads in the hardware-oriented method. In terms of performance, the average I/O RT was considered as a performance measure, because each hardware component communicates with the others as soon as possible in an asynchronous system. In the case of hardware-oriented controllers, much more signals are related to each output signal because of the large size of the corresponding STGs, and thus the average I/O response time increases. Moreover, internal signals for solving CSC violations introduce additional delays. Figure A· 1(e) in the Appendix shows that an inserted internal signal causes the delay of an output signal transition. In contrast with hardware-oriented controllers, process-oriented controllers are very compact and do not experience CSC violations, as shown in Table 2. Therefore, average I/O RTs of process-oriented control units are much smaller than those of hardware-oriented control units, as shown in Table 3. In the case of synthesis time, the synthesis of process-oriented controllers can be performed very quickly because of the small size of the corresponding STGs and inherent satisfaction of CSC property.

Note here that results of Table 3 are not limited to several benchmarks only because the results in Table 3 are based on the results in Tables 1 and 2. As previously mentioned, controllers in Tables 1 and 2 are basic and general components constituting a hardware-oriented and a process-oriented control units, and thus most of the process-oriented and hardware-oriented control units, including the control units in Table 3 consist of controllers in Tables 1 and 2. Therefore, for the other system benchmarks, as well as those in Table 3, we can expect similar data to that shown in Table 3. Consequently, although we did not verify the advantages of our process-oriented method over previous methods theoretically, we believe that our claim is general enough.

In the third experiment, we implement a '*differential equation solver*' at the structural VHDL code level based on the suggested method in order to check its efficiency and correctness by comparing it with human designed counterpart [19]. In [19], 4 distributed con-

**Table 3** Hardware-Oriented control units† v.s. Process-Oriented control units‡.

| name | lits | area | average I/O RT | H/W alloc. + | × | reg. | synthesis time (sec.) |
|---|---|---|---|---|---|---|---|
| †3FIR | 240 | 638.93 | 2.46 ns | 1 | 2 | 3 | 278.3 |
| ‡3FIR | 85 | 129.72 | 0.68 ns | 1 | 2 | 3 | 5.38 |
| †5FIR | 500 | 1216.34 | 2.59 ns | 2 | 2 | 5 | 1051.16 |
| ‡5FIR | 155 | 228.53 | 0.68 ns | 2 | 2 | 5 | 48.42 |
| †2IIR | 410 | 997.28 | 2.38 ns | 2 | 2 | 4 | 289.2 |
| ‡2IIR | 137 | 204.24 | 0.69 ns | 2 | 2 | 4 | 28.8 |
| †3IIR | 636 | 1429.06 | 2.62 ns | 2 | 2 | 6 | 711.2 |
| ‡3IIR | 184 | 298.34 | 0.68 ns | 2 | 2 | 6 | 69.92 |

Unit of area is 2-input NAND gate.

**Table 4** A controller comparison between [19]† and the Process-Oriented method‡ for the '*differential equation solver*.'

| name | lits | area | average I/O RT | synthesis time (sec.) |
|---|---|---|---|---|
| †$\text{AFSM}_{\text{ALU1}}$ | 43 | 65.86 | 1.37 ns | 6.3 |
| †$\text{AFSM}_{\text{ALU2}}$ | 139 | 200.0 | 2.49 ns | 20.1 |
| †$\text{AFSM}_{\text{MUL1}}$ | 42 | 64.16 | 1.82 ns | 3.4 |
| †$\text{AFSM}_{\text{MUL2}}$ | 15 | 23.94 | 1.32 ns | 2.5 |
| Total | 239 | 353.96 | - | 32.3 |
| ‡USC | 8 | 18.29 | 0.52 ns | 0.99 |
| ‡$\text{CNC}_{\text{while}}$ | 27 | 68.18 | 1.57 ns | 1.25 |
| ‡PSC3 | 9 | 17.30 | 0.44 ns | 1.27 |
| ‡PSC7 | 21 | 28.94 | 0.36 ns | 29.7 |
| ‡PC | 15×9 | 21.62×9 | 0.71 ns | 1.2 |
| Total | 200 | 327.29 | - | 34.41 |

Unit of area is 2-input NAND gate.

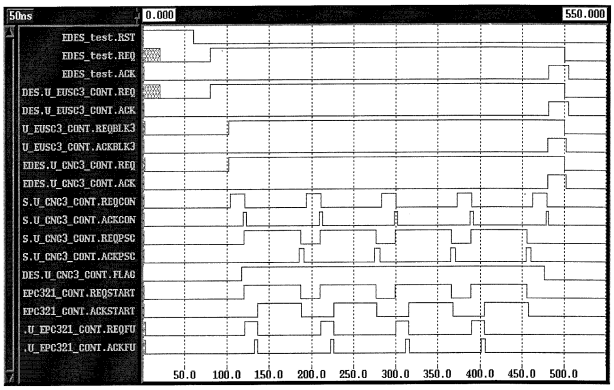



**Fig. 14** Simulation result I for a '*differential equation solver.*'

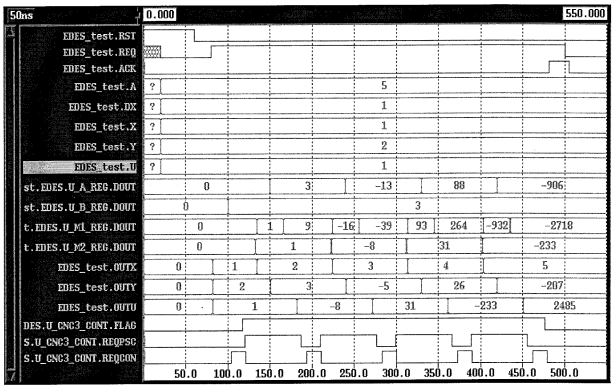



**Fig. 15** Simulation result II for a '*differential equation solver.*'

trollers are used and they are described in AFSM form. These four AFSM specifications were described very efficiently by a skilled designer. We synthesized these AFSM specifications using a burst mode synthesis tool, 3D [20]. Then we implemented another asynchronous control unit based on the process-oriented method. Table 4 shows the analytical data for those two implementations. It is worth noting that the results in Table 4 were obtained under different delay assumptions, as 3D is based on a burst mode while ours is based on a speed-independent assumption. In general, asynchronous circuits based on the burst mode delay assumption show better performance but less robustness than SI circuits, if the same controller generation method is adopted. More specifically, the most common SI circuit synthesis method uses C-elements but the 3D burst mode synthesis method does not use any latch. Consequently the experimental results in Table 4 show that a process-oriented approach can generate asynchronous control units, which are better than or comparable to human designed control units. Figures 14 and 15 present simulation results of the '*differential equation solver*' we implemented at VHDL level using SYNOPSYS and a 0.6 $\mu$m cell library. As Figs. 14 and 15 show, the process-oriented method based controllers work together using a 4-phase handshaking protocol. In order to satisfy the timing constraints given in Sect. 5, we inserted proper delays. In particular, for functional units and registers, 1.1–1.2 times bigger delays than their worst case delays were inserted.

From these three experiments we concluded that our approach presents a good and practical method of automatic asynchronous control unit generation.

## 7. Conclusion and Future Work

We have presented an automatic process-oriented control unit generation method. The proposed method has the following noticeable features;
- to present a systematic and hierarchical way to generate an asynchronous control unit which is separated into 'execution controllers' and 'execution order controllers' completely,
- to produce STGs satisfying four properties for speed-independent circuit implementation without any modification,
- to produce a hierarchically distributed asynchronous control unit standing at advantage in the points of area, performance, implementability and synthesis time due to the above two features.

Moreover, since all the procedures to derive asynchronous control units from a CDFG are suggested in an algorithmic and systematic way, they can be automated. Consequently our method is expected to present a good and practical way to generate asynchronous control units automatically as a part of asynchronous high-level synthesis.

This work has been performed as a part of building an asynchronous high-level synthesis CAD tool. In order to construct a complete CAD tool, researches about scheduling, resource allocation, resource binding and asynchronous architecture for an asynchronous system should be performed.

## Acknowledgment

This work has been supported in part by the Korea Science and Engineering Foundation under grant 20006-302-01-2 and by the KAIST/K-JIST IT-21 Initiative in BK21 of Ministry of Education.

### References

[1] S. Hauck, "Asynchronous design methodologies: An overview," Proc. IEEE, vol.83, no.1, pp.69–93, 1995.
[2] E. Pastor, J. Cortadella, A. Kondratyev, and O. Roig, "Structural methods for the synthesis of speed-independent circuits," IEEE Trans. Comput.-Aided Des. Integrated Circuits & Syst., vol.17, no.11, pp.1108–1129, 1998.
[3] S.M. Nowick, "Automatic synthesis of burst-mode asynchronous controllers," Ph.D. Thesis, Stanford University, 1995.
[4] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev, "Petrify: A tool for manipulating con-

current specifications and synthesis of asynchronous controllers," IEICE Trans. Inf. & Syst., vol.E80-D, no.3, pp.315–325, March 1997.
[5] T. Miyamoto and S. Kumagai, "On deriving logic functions of asynchronous circuits by STG unfoldings," IEICE Trans. Inf. & Syst., vol.E80-D, no.3, pp.336–343, March 1997.
[6] S.B. Park and T. Nanya, "Synthesis of asynchronous circuits from signal transition graph specifications," IEICE Trans. Inf. & Syst., vol.E80-D, no.3, pp.326–335, March 1997.
[7] E.M. Sentovich, K.J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P.R. Stephan, R.K. Brayton, and A. Sangiovanni-Vincentelli, "SIS: A system for sequential circuit synthesis," Technical Report UCB/ERL M92/41, U.C. Berkeley, May 1992.
[8] D.D. Gajski, N.D. Dutt, A.C.-H. Wu, and S.Y.-L. Lin, High-Level Synthesis: Introduction to Chip and System Design, Kluwer Academic Publishers, 1991.
[9] G. DeMicheli, Synthesis and Optimization of Digital Circuits, McGraw-Hill, 1994.
[10] J. Cortadella and R.M. Badia, "An asynchronous architecture model for behavioral synthesis," Proc. European Conference on Design Automation, pp.307–311, March 1992.
[11] R.M. Badia, J. Cortadella, E. Pastor, and A. Pardo, "A high-level synthesis system for asynchronous circuits," Sixth International Workshop on High-Level Synthesis, pp.87–94, Nov. 1992.
[12] I. Blunno and L. Lavagno, "Automated synthesis of micropipelines from behavioral verilog HDL," Proc. Sixth International Symposium on Advanced Research in Asynchronous Circuits and Systems, pp.84–92, April 2000.
[13] K.V. Berkel, "Handshake circuits. An asynchronous architecture for VLSI programming," International Series on Parallel Computation 5, Cambridge University Press, 1993.
[14] T. Kolks, S. Vercauteren, and B. Lin, "Control resynthesis for control-dominated asynchronous designs," Proc. Second International Symposium on Advanced Research in Asynchronous Circuits and Systems, pp.233–243, March 1996.
[15] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev, "A region-based theory for state assignment in speed-independent circuits," IEEE Trans. Comput.-Aided Des. Integrated Circuits & Syst., vol.16, no.8, pp.793–812, 1997.
[16] T.A. Chu, "Synthesis of self-timed VLSI circuits from graph-theoretic specifications," Ph.D. Thesis, MIT, June 1987.
[17] A. Kondratyev, M. Kishinevsky, B. Lin, P. Vanbekbergen, and A. Yakovlev, "Basic gate implementation of speed-independent circuits," Proc. ACM/IEEE Design Automation Conference, pp.56–62, June 1994.
[18] T. Murata, "Petri nets: Properties, analysis and applications," Proc. IEEE, vol.77, no.4, pp.541–580, 1989.
[19] K.Y. Yun, P.A. Beerel, V. Vakilotojar, A.E. Dooply, and J. Arceo, "The design and verification of a high-performance low-control-overhead asynchronous differential equation solver," IEEE Trans. VLSI Systems, vol.6, no.4, pp.643–655, 1998.
[20] K.Y. Yun and D.L. Dill, "Automatic synthesis of extended burst-mode circuits: Part II (Automatic synthesis)," IEEE Trans. Comput.-Aided Des. Integrated Circuits & Syst., vol.18, no.2, pp.118–132, 1999.

## Appendix

Complete State Coding (CSC) property is a necessary condition for hazard-free speed-independent asynchronous control circuit implementation. Its formal definition is as follows;

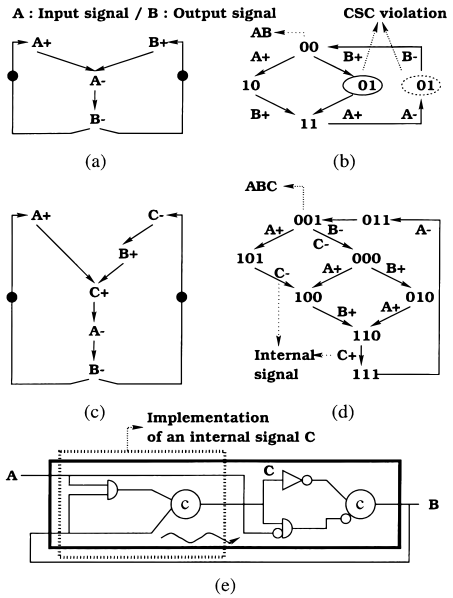



**Fig. A· 1** (a) An STG with a CSC violation, (b) An SG for the STG in (a), (c) An STG satisfying CSC property, (d) An SG for the STG in (c), (e) An asynchronous control circuit for the STG in (c).

**Definition 6:** An STG is said to satisfy Complete State Coding (CSC) property iff, when the same binary code is assigned to different markings, the transitions of the non-input signals enabled at both markings are identical, i.e.

$$\forall \mathsf{M_i}, \mathsf{M_j} \in [\mathsf{M_0}\rangle : \mathsf{i} \neq \mathsf{j} \wedge \lambda(\mathsf{M_i}) = \lambda(\mathsf{M_j}) \Rightarrow (\forall \mathsf{s} \in \mathsf{S_{ni}}, \exists \mathsf{s_i}*, \mathsf{s_j}* : \mathsf{M_i}[\mathsf{s_i}*\rangle \Leftrightarrow \mathsf{M_j}[\mathsf{s_j}*\rangle)$$

$[\mathsf{M_0}\rangle$ represents the set of markings reachable from an initial marking $\mathsf{M_0}$, $\lambda(\mathsf{M})$ denotes a binary code which corresponds to current values of signals in a marking $\mathsf{M}$, $\mathsf{S_{ni}}$ is the set of non-input signals, and $\mathsf{s_i}*$ is the i-th $\mathsf{s_i}+$ or $\mathsf{s_i}-$ of a signal $\mathsf{s}$. $\mathsf{M}[\mathsf{s_i}*\rangle$ means that a signal transition $\mathsf{s_i}*$ is enabled in the marking $\mathsf{M}$.

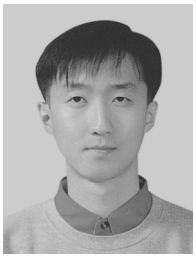
In general, CSC violations can be solved by inserting internal signals. Figure A· 1 is an example to show how to transform and implement an STG with CSC violations into an asynchronous control circuit. Let 'A' be input and 'B' be output signals for the STG and the corresponding SG in Figs. A· 1(a) and (b). As shown in Fig. A· 1(b), there exist two different markings with the same binary code '01' but with the different set of non-input enabled signals. Therefore, the STG in Fig. A· 1(a) does not satisfy CSC property. Since CSC property is a necessary condition for hazard-free speed-independent asynchronous control circuit implementation, we should make the STG to satisfy CSC property. Figures A· 1(c) and (d) show the STG and the corresponding SG satisfying CSC property and it is derived

from the STG in Fig. A· 1(a) through inserting an internal signal 'C.' Since the STG in Fig. A· 1(c) satisfies CSC property, it can be synthesized into an asynchronous control circuit shown in Fig. A· 1(e).

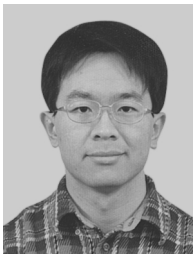
Although CSC violations can be solved by inserting internal signals, CSC violations are still very critical in an asynchronous control circuit synthesis procedure because of following reasons;

- It is very time-consuming to solve CSC violations.
- Internal signal insertions make an original STG more complex and larger and thus increases its synthesis time rapidly.
- Since synthesis is performed for output and internal signals, area of a resulting circuit increases. For example, Fig. A· 1(e) shows that half of the circuit is used for implementing internal signal 'C.'
- Internal signals increase delay of a resulting control circuit. As shown in Fig. A· 1(e), internal signals are located between input and output ports of a controller and thus they cause increase of the controller's delay.

Therefore, it is one of the best ways to area and speed efficient asynchronous control circuit synthesis to describe an STG satisfying CSC property initially.



**Euiseok Kim** received the B.E. in the Department of Computer Science from Yonsei University, Korea and M.S. in the Department of Information and Communications of Kwangju Institute of Science and Technology (K-JIST), Korea in 1995 and 1997, respectively. Currently he is working toward Ph.D. degree in the department of information and communications, K-JIST. His research interests include Petri Net theory and its applications to concurrent systems, asynchronous circuit and system design, and computer-aided design. He is a student member of ACM, IEEE and KISS.



**Jeong-Gun Lee** received the B.E. in the Department of Computer Science from Hallym University, Korea and M.S. in the department of information and communications of Kwangju Institute of Science and Technology, Korea in 1996 and 1998, respectively. Currently he is working toward Ph.D. degree in the department of information and communications, K-JIST. His research interests include computer architecture, asynchronous circuit, concurrent systems, and parallel/distributed computing. He is a student member of IEEE and KISS.

**Dong-Ik Lee** received the B.E. from Yeungnam University, Korea, M.E. and Dr.Eng. from Osaka University, Japan, in 1985, 1989 and 1993, respectively. He was a research associate in the department of electronic engineering of Osaka University from 1990 to 1995. From 1993 through 1994, he was a visiting assistant professor in coordinated science Lab. of University of Illinois. He is currently an associate professor in the department of information and communications in Kwangju Institute of Science and Technology from 1995. His research interests include Petri Net theory and its applications to concurrent systems, asynchronous circuits design, computer-aided design and agent systems.