$ADF(\infty)$ is also equivalent to EF

Yasuo MATSUBARA

ABSTRACT: In this paper, the ability of the class $ADF(\infty)$, where an arc can hold any number of tokens, is investigated and is shown to be equivalent to EF as similar to the class $ADF(1)$ where an arc can hold at most one token. Further, the class $DF_{\pi}(\infty)$is shown to have an ability greater than or equal to that of the class $DF_{arb}(\infty)$.

1 Introduction

The dataflow language is a programming language describing parallel processing based upon a flow of the data, and is entirely different from the conventional languages that have positively described order control in some meanings[1][2][3]. It is quite interesting to discern what kind of expression ability the language based upon such a principle possess as a programming language. In these days, comaprison of programming languages are made by considering the class of functionals realized by program schemata[4][5]. A program schema becomes a program when an interpretation is given. An interpretation gives a data domain, and gives a function for each function symbol and gives a predicate for each predicate symbol.

As the descriptive process to now, it is at first shown that the class DF of a simple dataflow schemata has the expression ability equivalent to the class EF of effective functionals in the total interpretaions[6]. In total interpretation, all the functions and predicates given to each function symbols and predicate symbols are totally defined on the data domain. In real programming environment, however, functions or predicate which should be realized by subroutines or functions are not necessarily totally defined. Therefore, expressive power of program schemata should be compared in partial interpretaions, where functions or predicates can be partially defined. In partial interpretation, it is shown that the class DF is equivalent to the class of deterministic effective functionals EF^d which is a proper subclass of EF[7]. This means that the class DF has the expressive power which can be realized by the class of sequential program schemata.

To show that the class of dataflow schemata is powerfull than the class of sequential program schemata, the class ADF of dataflow schemata is proposed and is shown to have the expressive power equivalent to the class EF in partial interpretations[8]. To investigate the expressive power of the class of dataflow schemata, it is important to distinguish between cases where an arc can hold any number of tokens and the cases where an arc can hold one token at most. To distinguish these cases explicitly, we append(μ) after the class name. $\mu=1$ means that an arc hold one token at most, and $\mu=\infty$ means that an arc can hold arbitrary numbers of tokens and can be considered as *FIFO* queue. Using this notation, Jaffe's *DF* is written as $DF(\infty)$, and Matsubara and Noguchi's *ADF* should be written as $ADF(1)$. *ADF* is stregthend with two kind of capabilities than the class *DF*. One is the arbiter and the other is the open recursive procedure. Therefore, it is problematic whether the both capabilities are necessary for *ADF* to be equivalent to *EF*. To answer the question, the classes having only one capabily have been investigated. However, it was difficult to decide whether the class $DF_{arb}(\infty)$, which utilizes the arbiter solely, incudes the class $DF_{arb}(1)$ or not. Therefore, we have introduced the new class $DF_{\pi}(\infty)$ utilizing a new device π-gate instead of the arbiter[11].

In this paper, we investigate the expressive power of the class $ADF(\infty)$ and show that the class is also equivalent to *EF* in partial interpretation. Further, we investigate the relation between $DF_{\pi}(\infty)$ and $DF_{arb}(\infty)$, and give the relation $DF_{\pi}(\infty)\geqq DF_{arb}(\infty)$.

2 Program Schemata

The program schema (hereafter just called schema) includes finite types of function symbols and finite types of predicate symbols, and the number of the arguments of the individual symbols is settled. In this paper, the function symbols are taken from the set $\mathcal{F}=\{F1,F2,...\}$ and the predicate symbols are taken from the set $\mathcal{P}=\{P1,P2,...\}$. Rf and Rp expresses the numbers of the arguments of f and p respectively. For the sake of simplicity, let $RFi\geqq 1$ and $RPi\geqq 1$ with respect to arbitrary $i\geqq 1$. On the other hand, variables are taken from the set $\mathcal{X}=\{X1,X2,...\}$.

When interpretation is given, the schema becomes a concrete program. The program gives at most one calculation result by executing concrete calculation when input values are provided.

DEFINITION 2.1

We assume that the set of the function symbols and predicate symbols included in the schema S are repectively $\mathbf{F}=\{f_1,f_2,...,f_m\}$ and $\mathbf{P}=\{p_1,...,p_n\}(\mathbf{F}\subset\mathcal{F},\mathbf{P}\subset\mathcal{P})$. Then the interpretation I for the schema S gives the data domain D, gives a map from D^{Rf} to D for each function symbol f, and gives a map from $D^{Rf}$to truth-and-falsity value $\{F,T\}$ for each predicate symbol p.

When all of the maps given by the interpretation are defined on whole region of the domain D, the interpretation is called the total one. The partial interpretation is an interpretation which give maps not necessarily totally defined. Hereunder, interpretation means an partial one.

DEFINITION 2.2

We assume that two schemata $S1$ and $S2$ have simultaneously the set $\mathbf{X} \subset \mathcal{X}$ of the input variables, the set $\mathbf{F} \subset \mathcal{F}$ of the function symbols, and the set $\mathbf{P} \subset \mathcal{P}$ of the predicate symbols. Provided that $S1$ and $S2$ respectively output the same value or do not respectively output at the same time when interpretation I is given, $S1$ and $S2$ are said to be equivalent under I. On the other hand provided that they are equivalent under arbitrary interpretation, they are said to be just equivalent.

When there exists a schema of the class β equivalent to the individual schema of the class α, it can be written that $\alpha \leqq \beta$ and β is said to include α. When $\alpha \leqq \beta$ and $\beta \leqq \alpha$, this relation is expressed as $\alpha \equiv \beta$ and the two classes are said to be equivalent. When $\alpha \leqq \beta$ is satisfied and there is a schema of β which cannot be simulated by one of α, it is written as $\alpha < \beta$ and β is said to properly include α.

In the next stage, let the computation composing an effective functional be defined.

DEFINITION 2.3

(a) Let $\mathbf{X} \subset \mathcal{X}$ be a finite set of the input variables, let $\mathbf{F} \subset \mathcal{F}$ be a finite set of the function symbols, and let $\mathbf{P} \subset \mathcal{P}$ be a finite set of the predicate symbols. In this occasion, let Λ and Π be the minimum set sufficing the following conditions.

I. $\mathbf{X} \subset \Lambda$

II. For each $f \in \mathbf{F}$ and $e_1, \ldots, e_{Rf} \in \Lambda$,

$f(e_1, \ldots, e_{Rf}) \in \Lambda$

III. For each $p \in \mathbf{P}$ and $e_1, \ldots, e_{Rp} \in \Lambda$,

$p(e_1, \ldots, e_{Rp}) \in \Pi$ and $\neg p(e_1, \ldots, e_{Rp}) \in \Pi$

where $\neg$ is the symbol expressing the negation.

The elelments of Λ is called an expression concerning **X** and **F**, whereas the elements of Π is called a proposition concerning **X**, **F** and **P**.

(b) The computation is a sequence comprised of an expression and proposition, and is finally terminated with the expression. Therefore, the elements of $(\Lambda \cup \Pi)^*\Lambda$ are the computations.

When the interpretation I is given and the elements of D are given to the each input variables, the value of the computation is defined as shown below. That is to say, if and only if, the expressions and propositions composing the computation are all defined and the propositions have all true values, the computation has a value. The said value is the one of the final expression.

DEFINITION 2.4

The effective functional S, an element of the class EF, is defined by the 4 - tuple $S = \langle \mathbf{X}, \mathbf{F}, \mathbf{P}, \mathcal{T} \rangle$. This is a recursively enumerable set of computations.

Here,

(1) Let $\mathbf{X} \subset \mathcal{X}$, $\mathbf{F} \subset \mathcal{F}$, and $\mathbf{P} \subset \mathcal{P}$ be finite sets of the input variables, function symbols, and predicate symbols.

(2) $\mathcal{T}$ indicate the Turing machine, and outputs the i-th calculation $\mathcal{T}(i)$ with the positive integer i as an input under proper coding.

(3) Provided that the computation for more than 2 inputs, e.g. $\mathcal{T}(i)$ and $\mathcal{T}(j)$ has values when interpretation and input are given, let it be assumed that they are the same values. The said value is the value of S.

3 Dataflow Schemata

The dataflow schema is a kind of the program schema, but the way of the execution is made in parallel in accordance with the flow of the data.

In the class of dataflow schemata, it is necessary to distinguish the case where an arc can hold at most a token and the case where an arc can hold arbitrary numbers of tokens.

Both classes are equal from a viewpoint of the syntax. When the arc holds arbitrary number of the tokens, the order of the tokens is maintained. That is to say, the arc plays a role of FIFO.

Definition 3.1

The schema belonging to the class *adf*(∞) or *adf*(1) is a finite sequence generated from ⟨Schema⟩ in accordance with the grammar shown below.

⟨Schema⟩ ::= Program ⟨SchemaName⟩ (y^d:x_1^d,..,x_n^d); ⟨MainBody⟩ ; ⟨DeclarationPart⟩

⟨MainBody⟩ ::= ⟨Body⟩

⟨DeclarationPart⟩ ::= ε | ⟨Declaration⟩

| ⟨Declaration⟩ ⟨DeclarationPart⟩

⟨Declaration⟩ ::= Procedure ⟨ProcedureName⟩ (y_1,...,y_r:x_1,...,x_s) ⟨Body⟩ ;

⟨Body⟩ ::= ⟨InitialSetting⟩ ; begin ⟨StatementSequence⟩ end

⟨StatenmetSequence⟩ ::= ⟨Statement⟩ | ⟨Statenment⟩ ; ⟨StatementSequence⟩

⟨Statenment⟩ ::=

$y^d = f(x_1^d,\ldots,x_{Rf}^d)$ | $y^c = p(x_c^d,\ldots,x_{Rf}^d)$

| $y^d = T^d(v^c,x^d)$ | $y^d = F^d(v^c,x^d)$ | $y^d = M^d(v^c,x_1^d,x_2^d)$

| $y^c = T^c(v^c,x^c)$ | $y^c = F^c(v^c,x^c)$ | $y^c = M^c(v^c,x_1^c,x_2^c)$

| $(y_1^d,\ldots,y_r^d) = L^d(x^d)$ | $(y_1^c,\ldots,y_r^c) = L^c(x^c)$

| $y^c = x_1^c \cdot x_2^c$ | $y^c = x_1^c \vee x_2^c$ | $y^c = \neg x^c$:

| $(y_1,\ldots,y_r)$ = ⟨ProcedureName⟩ $(x_1,\ldots,x_s)$

| $y^d = \mathrm{Arb}(x_1^d,x_2^d)$

⟨InitialSetting⟩ ::= init ⟨SettingSequence⟩ end

⟨SettingSequence⟩ ::= ⟨SettingSetatement⟩

⟨SettingSetatement⟩ ; ⟨SettingSequence⟩
⟨SettingSetatement⟩ : : $= y^c := T | y^c := F$

In the above, let n, r, $s \geqq 1$, y^d, x^d, etc. indicate data arcs. y^c, v^c, x^c, etc. denote comtrol arcs. y, x, etc. represent data arcs or control arcs. Let the arcs be expressed by the elements of the set $\mathcal{X}$.

With (y^d:x^d_1,...,x^d_n) after ⟨SchemaName⟩ ,y^d shows the output arc of the schema, whereas x^d_1,...,x^d_n is shown to be the input arc of the schema. x^d_1,...,x^d_n are concurrently the input variables.

On the declaration of procedure, y_1,...,y_r on the left of ':' show the output arcs from the procedure and x_1,...,x_s on the right of ':' show the input arcs to the procedure.

The statements including T, F, M, and L in their right-hand side show respectively T-gate, F-gate, merge, and link. The superscripts d and c on the right shoulders show that they stands for data token and control token as operation objective, respectively. When the matter common to these is discussed, 'd' and 'c' are sometimes omitted. The link has more than one piece of the output arc.

The statements including ⟨ProcedureName⟩ in the right-hand side show a call of the procedure. x_1,...,x_s are the input arcs to the procedure, whereas y_1,...,y_r are the output arcs from the procedure. This ⟨ProcedureName⟩ is declared in the declaration part, and let the number and type of the input/output arcs be in agreement with the declaration time.

The statements including 'Arb' show the arbiter.

f indicates the function symbol taken from $\mathcal{F}$, whereas p denotes the predicate symbol taken from $\mathcal{P}$.

The arc appearing in a schema appears precisely one time on the output arc or in the right hand side of a statement in the schema, and appears precisely one time as the input arc or in the left-hand side of a statement in the schema.

The individual statements referred to above are illustrated in Fig. 3.2. The data arc is depicted by the arrow as Fig.3.1(a), whereas the control arc is sketched by arrow as Fig.3.1 (b). The arrow of Fig.3.1(c) exhibits either of the data arc or control arc. The initial setting is exhibited by ● as Fig3.1(d) on the arc.

In Fig.3.2(o), x′ , y′ , etc. are the arcs at the time of the declaration. The input arcs and the output arc of the schema, are clearly depicted as Fig.3.2(p), (q).

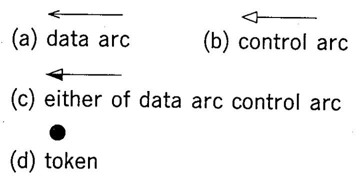


Fig. 3.1 Some depictions of arcs and token.

In the description below, let the figure and statement be used according to requiement on the assumption that they are equivalent to each other.

Definition3.2

The semantics of *adf*(∞) : the schema of *adf*(∞) is driven into an action as shown below when interpretation I is given and input data are provided to the individual input variables.

On the control arc, arbitrary pieces of truth-and-falsity values are placed. On the data arc, arbitrary pieces of the elements of the data domain D given by the interpretation I are placed. The truth-and-falsity values and the elements of D on these arcs are called tokens, and is shown by ● in the figure.

In the initial state, the truth-and-falsity values shown by initial setting are placed on the corresponding control arcs. On the other hand, let it be assumed that the input data given by the individual input variables are placed on the individual input arcs without any rectification. The other arcs are left empty.

Let it be assumed that the execution is made on the discrete time such as $t=0,1,2,...$

On the assumption that u is a predicate symbol or function symbol and on the supposition that $d_1^d,...,d_{Ru}^d \in D$, $\tau(u,d_1^d,...,d_{Ru}^d)$ gives the computation time of $u(d_1^d,...,d_{Ru}^d)$. A time function τ that gives infinity computation time when the value of $u(d_1^d,...,d_{Ru}^d)$ becomes undefined and that gives the computation time of positive integer values when the value is defined in the interpretation I is called a time function consistent with the interpretation I.

In the next stage, the actions of the individual statements are described.

$y=u(x_1^d,...,x_{Ru}^d)$, where let u be a function symbol or predicate symbol:

When there exists a token sequence that is not empty on the individual input arcs at the time $t=\tau_0$ and its head is $d_1,...,d_{Ru}$, the head token on the individual input arcs is removed and the token as the computation result is placed on the tail of the token sequence on the output arc y at the time $t=\tau_0+\tau(u,d_1^d,...,d_{Ru}^d)$.

$y=T(v^c,x)$: When T is placed on the head of the token sequence on v^c and the token is placed on x at $t=\tau_0$, the token on the head on v^c is removed at $t=\tau_0+1$. Furthermore it is noticed that the token on the head of x is transferred to the tail of the token sequence on y.

On the other hand, if F is placed on the head of v^c and the token is placed on x at $t=\tau_0$, it is noticed that not only the token on the head of v^c but also the token on the head of x are removed at $t=\tau_0+1$.

The action of $y=F(v^c,x)$ is obtained by inverting the truth-and-falsity value of the token on v^c.

$y=M(v^c,x_1,x_2)$:

When T is placed on the head of v^c at $t=\tau_0$ and token is placed on x_1, it is noticed that the token on the head of v^c is removed and the token on the head of x_1 is transferred to the tail of the token sequence on y at $t=\tau_0+1$.

On the other hand, if F is placed on the head of v^c at $t=\tau_0$ and the token is placed on x_2, it is noticed that the token on the head of v^c is removed and the token on the head of

x_2 is transferred to the tail of the token sequence on y at $t = \tau_0 + 1$.

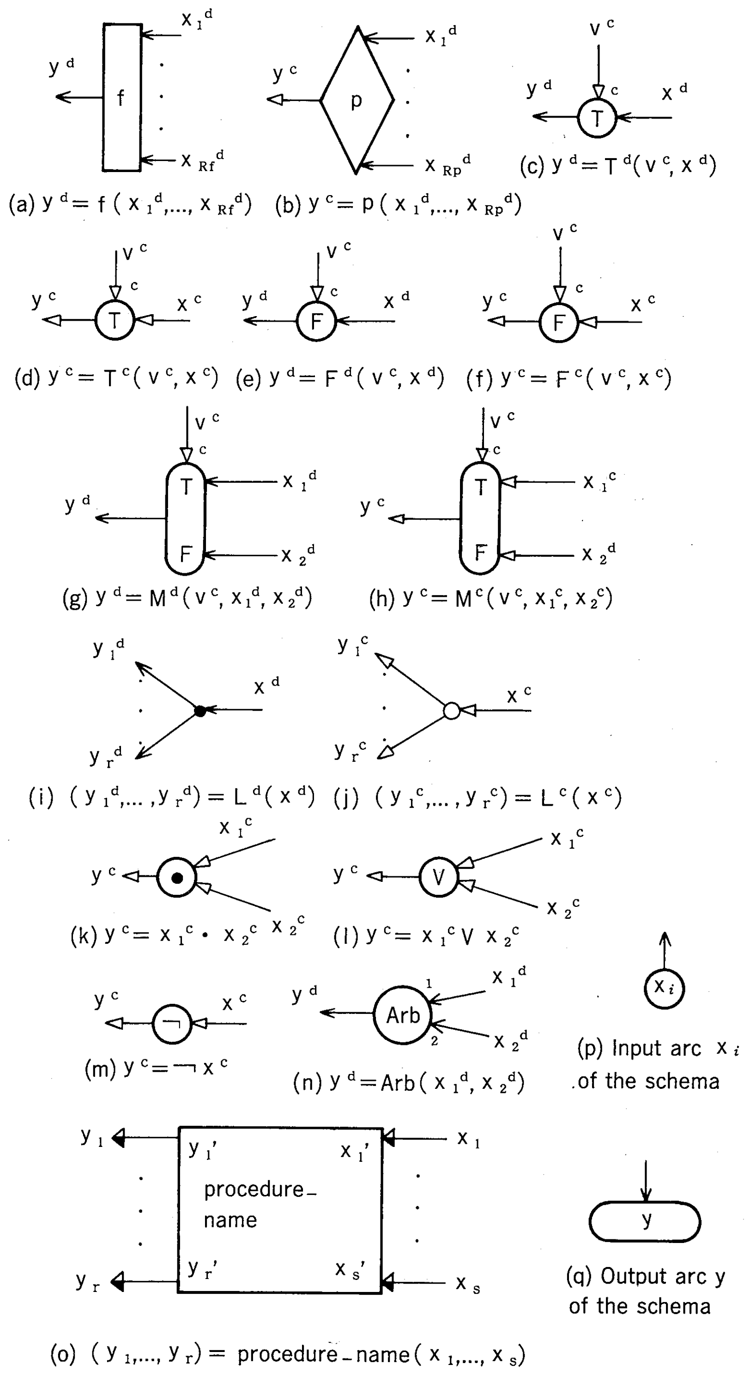

(a) $y^d = f(x_1^d, \ldots, x_{Rf}^d)$
(b) $y^c = p(x_1^d, \ldots, x_{Rp}^d)$
(c) $y^d = T^d(v^c, x^d)$
(d) $y^c = T^c(v^c, x^c)$
(e) $y^d = F^d(v^c, x^d)$
(f) $y^c = F^c(v^c, x^c)$
(g) $y^d = M^d(v^c, x_1^d, x_2^d)$
(h) $y^c = M^c(v^c, x_1^c, x_2^c)$
(i) $(y_1^d, \ldots, y_r^d) = L^d(x^d)$
(j) $(y_1^c, \ldots, y_r^c) = L^c(x^c)$
(k) $y^c = x_1^c \cdot x_2^c$
(l) $y^c = x_1^c \vee x_2^c$
(m) $y^c = \neg x^c$
(n) $y^d = \mathrm{Arb}(x_1^d, x_2^d)$
(p) Input arc x_i of the schema
(q) Output arc y of the schema
(o) $(y_1, \ldots, y_r) = \text{procedure\_name}(x_1, \ldots, x_s)$
procedure_name

Fig. 3.2 Graphical notation
for each statement

$y^c = x_1^c \cdot x_2^c$, $y = x_1^c \vee x_2^c$, $y^c = \neg x^c$:

When the token is placed on the individual input arcs at $t = \tau_0$, the token on the head of the individual input arc is removed at $t = \tau_0 + 1$. Thus also it is noticed that a token having a value of result of the logical operation is placed on the tail of the token sequence on the output arc at that time.

$(y_1^d,\ldots,y_r^d) = L^d(x^d)$, $(y_1^c,\ldots,y_r^c) = L^c(x^c)$:

When token is placed on the input arc at $t = \tau_0$, it is noticed that the token on the head of the input arc is removed and the tokens having the same values are placed on the tail of the token sequence on the individual output arcs at $t = \tau_0 + 1$.

$y^d = \mathrm{Arb}(x_1^d, x_2^d)$:

When token is placed on x_1^d at $t = \tau_0$, it is noticed that the token on the head is transferred to the tail of the token sequence on y^d at $t = \tau_0 + 1$.

On the other hand when no token is placed on x_1^d and a token is placed on x_2^d at $t = \tau_0$, it is noticed that the token on the head of x_2^d is, at $t = \tau_0 + 1$, transferred to the tail of the token sequence on y^d.

$(y_1,\ldots,y_r) = \langle\mathrm{ProcedureName}\rangle\ (x_1,\ldots,x_s)$:

In lieu of this procedure call, the body part of the declared procedure is expanded. However, let it be understood that the arc name will be systematically changed at that time so that an arc with the same name will not come into existence.

Let it be understood that the input/output arc at the time of declaration will be replaced with the arc at the time of call.

Let it be understood that the value of the token placed for the first time onto the output arc in the schema will be the output result of the schema.

Definition 3.3 Semantics of adf(1):

When interpretation *I* and input values are given, the schema of adf(1) is fundamentally driven into action the same as in the case of the schema of adf(∞) except that a single piece of token at most can only be placed on the arc.

In the description hereunder, the action of the individual statements is explained.

$y = u(x_1^d,\ldots,x_{Ru}^d)$, where let u be a function symbol or predicate symbol:

Time function τ is as same as in the Definition 3.2.

When tokens $d_1,\ldots,d_{Ru}$ are placed on the individual input arcs and concurrently no tokens are placed on y at the time $t = \tau_0$, it is noticed that at $t = \tau_0 + \tau(u, d_1^d,\ldots,d_{Ru}^d)$, the tokens on the individual input arcs are removed and the token of the computation result is placed on the output arc y.

$y = T(v^c, x)$:

When *T* is placed on v^c at $t = \tau_0$ and token is placed on x and concurrently no token is placed on y, the token on v^c is removed at $t = \tau_0 + 1$. Thus it is noticed that the token on x is transferred onto y.

When *F* is placed on v^c at $t = \tau_0$ and token is placed on x and concurrently no token is placed on y, the token on x and the token on v^c are removed at $t=\tau_0+1$. Thus it is noticed that y remains empty.

The action of $y = F(v^c,x)$ is the one obtained by reversing the truth-and-falsity value of the token on v^c.

$y=M(v^c,x_1,x_2)$:

When T is placed on v^c and token is placed on x_1, and concurrently no token is placed on y at $t=\tau_0$, it is noticed that the token on v^c is removed and the token on x_1 is transferred on y at $t=\tau_0+1$.

On the other hand, if F is placed on v^c at $t=\tau_0$ and the token is placed on x_2 and concurrently no token is placed on y, it is noticed taht the token on v^c is removed and the token on x_2 is transferred on y at $t=\tau_0+1$.

$y^c=x_1^c \cdot x_2^c$, $y=x_1^c<x_2^c$, $y^c=\neg x^c$:

When tokens are placed on the individual input arcs and no token is placed on y at $t=\tau_0$, it is noticed that the tokens on the individual input arcs are removed and the token as the result of the operation is placed on the output arc.

$(y_1^d,...,y_r^d)=L^d(x^d)$, $(y_1^c,...,y_r^c)=L^c(x^c)$:

When a token is placed on the input arc and concurrently all the output arcs are empty at $t=\tau_0$, it is noticed that the token on the head is removed and the tokens having the same value are placed on the individual arcs.

$y^d=Arb(x_1^d,x_2^d)$:

When a token is placed on x_1^d at $t=\tau_0$ and no token is placed on y^d, it is noticed that the token on x_1^d is transferred onto y^d at $t=\tau_0+1$.

On the other hand when no token is placed on x_1^d and token is placed on x_2^d and concurrently no token is placed on y^d at $t=\tau_0$, it is noticed that the token on x_2^d is transferred to y^d at $t=\tau_0+1$.

$(y_1,...,y_r)=$ ⟨ProcedureName⟩ $(x_1,...,x_s)$:

In lieu of this procedure call, the body part of the declared procedure is expanded. Here, let the arc name be changed systematically so that no arc having the same arc name will come into existence. Let the input/output arc at the time of declaration be replaced with the one of the procedure call.

Let the value of the token to be placed on the output arc in the schema for the first time be the output result of the schema.

Definition 3.4 With respect to $\mu=1$ or ∞, $ADF(\mu)$ is the class comprised of the ones sufficing the 2 conditions shown below in the schema of the class $adf(\mu)$.

(1)When interpretation I and input value are given, the same result is provided for an arbitrary time function τ consistent with I.

(2) In the procedure declared recursively, there is no statement that can be driven into action in an initial state. (That is to say, no action is started until the token comes in from the other part. Thanks to this condition, the procedure can be expanded corresponding to

necessity.)

Definition 3.5 With respect to $\mu=1, \infty$, the schema of the class $DF(\mu)$ is comprised of the ones not including the arbiters, call and declaration of the procedure in the schema of the class $ADF(\mu)$.

Definition 3.6 With respect to $\mu=1, \infty$, the class $DF_{arb}(\mu)$ is comprised of the ones including no procedure call or no procedure declaration, in the schema of $ADF(\mu)$.

Definition 3.7 The class $DF_{\pi}(\infty)$ is the class givin by adding a π-gate as possible individual statement in the class $DF(\infty)$. The π-gate is a statement in style of $y^c=\pi(v^c,x)$.

On one hand, the token on the top of v^c is removed and F is added to the tail of y^c at the time $t=\tau_0+1$, provided that there exist a token on the input arc v^c and there is no token on x at the time $t=\tau_0$.

On the other hand, the token on the top of v^c and x are removed and T is added to the tail of y^c at the time $t=\tau_0+1$, provided that there exist a token on each of the input arcs at $t=\tau_0$.

Let the statement be illustrated as the depiction shown in Fig. 3.3.

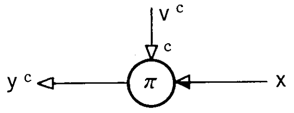


Fig. 3.3 $y^c=\pi(v^c, x)$

Arbiter and π-gate posseses a timing dependent property. That is, the value of the output token depends on which token is inputted faster.

Definition 3.8

(a)Boolean graph is an acyclic part of dataflow schema comprised exclusively of logical operaotors and control links of the form as follows:

$y^c=x_1^c \cdot x_2^c$, $y^c=x_1^c \vee x_2^c$, $y^c=\neg x^c$, $(y_1^c,\ldots,y_r^c)=L^c(x^c)$

(b)When a Boolean graph having r input arcs and s output arcs satisfy the following conditions(1) and(2), the graph is said to realize a Boolean function $g:\{F,T\}^r \rightarrow [F,T\}^s$.

(1)Until a token is inputted to each of input arcs, no token is outputted from any output arc. On the other hand, when a token is inputted to each input arc, a token is outputted from each output arc.

(2)Assume that the token inputted to i-th input arc is b_i and let the token outputted from j-th output arc is c_j. Then the following relation is satisfied:

$\langle c_1,\ldots,c_s\rangle = g(b_1,\ldots,b_r)$

For an arbitrary logical function, a Boolean graph realizing the function can easily be composed. As is shown in[6,7], any kind of finite state machine can be composed by connecting the output arcs to the input arcs of a Boolean graph realizing an appropriate logical function.

4 $EF \equiv ADF(\mu)$

The purpose of this chapter is to show that any schema $S \in EF$ can be simulated by a schema of $ADF(\infty)$. Firstly, we should develope a method to evaluate a i-th computation. When i is given, the schema of $ADF(\infty)$ should simulate a Turing machine to yield the i-th computation which should be evaluated. The simulation of the Turing machine is considered to be easier for the schema of $ADF(\infty)$ than for the schema of $ADF(1)$, because an arc can be used to simulate a tape of Turing machine. As a matter of fact, Jaffe had developed a method to simulate a Turing machine by the schema of $DF(\infty)$. On the contrary, for the schema of $ADF(1)$, other method of Turing machine's simulation was necessary. Matsubara and Noguchi had developed such a method. Here, we show that the same method is applicable for the schema of $ADF(\infty)$ to simulate the Turing machine. Hereunder, we consider $ADF(\mu)$ for the both cases $\mu=1$ and $\mu=\infty$ in parallel. The reader is expected to inspect the discussion in the bellow whether it is proper for both cases.

It is well known that a pair of pushdown stacks attached to the finite controller is enough to simulate a Turing machine with a tape of infinite length.

In Fig. 4.1(a), a recursive procedure DSTA for that purpose is shown. In this procedure, the part enclosed by a one-dot chain line appears repeatedly on a straight line. This part that is to be a unit is called a cell. This initial state shows the cell to be in an empty state. The fact that there exists the control token F on the part upper than the broken line means that t he part lower than the broken line is empty. Meanwhile the fact that there exists the control token T on the part upper than the broken line means that there exists a data token on the part lower than the broken line. The data token is maintained on the part of the circle drawn in broken line.

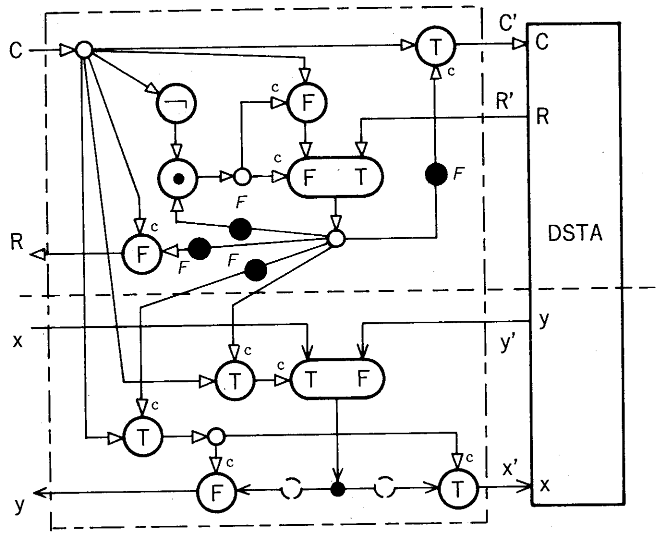
The action of this procedure is explained as shown below.

When a data token is pushed, let the control token T be inputted to C from the left, and let the data token desired to be pushed to x be inputted from the left.

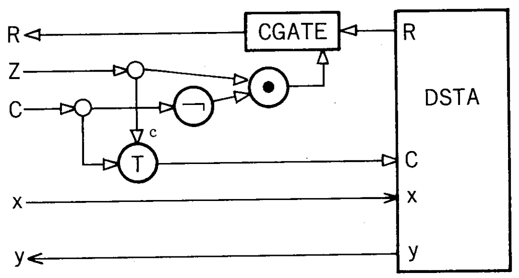
For the above explanation, the following 2 cases are in consideration.

(1) When the cell in question has been empty:

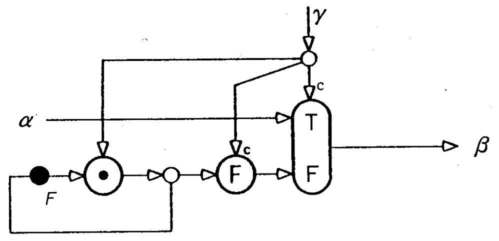
The control token on the upper side is changed from F to T and the data token is maintained in the lower side. Nonetheless, no token is outputted either to the left or to the right.



(a) Procedure DSTA(y,R:C,x)



(b) Procedure DSTACK(y,R:C,x)



(c) Procedure CGATE(β:α,γ)

Fig. 4.1 Construction of DSTACK.

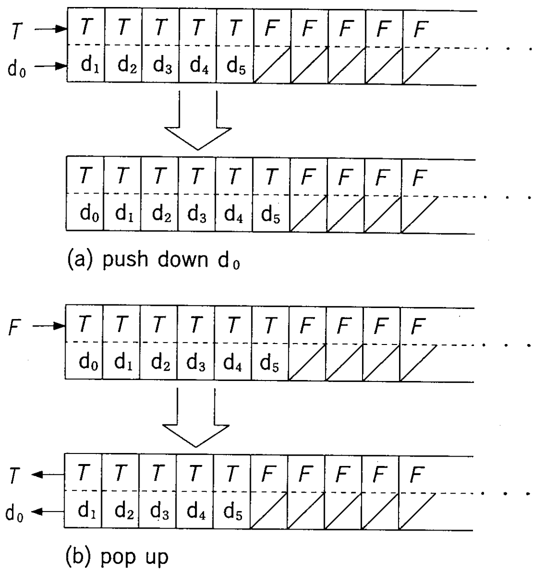


Fig. 4.2 Motion of DSTACK

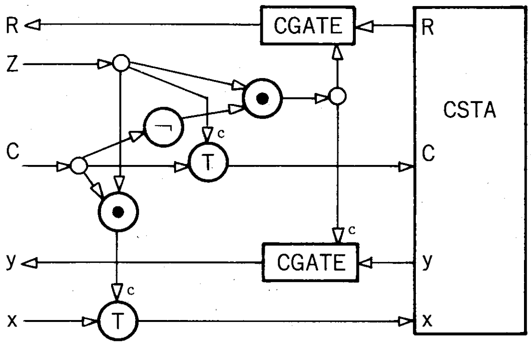


Fig. 4.3 Procedure CSTACK(R,y:Z,C,x)

(2) When the cell has been occupied with data tokens:

The control token on the upper side has been T and this value is unchanged. The data inputted from the left are maintained on the lower side, and the data token that has been maintained is outputted to the right. On the other hand, the control token T is outputted from C′ to the right.

On the other hand when the data token is desired to be popped up, let the control token F be inputted from C. With respect also to this, 2 cases are available.

(1) When the cell in question has been empty:

The control token F is outputted from R to the left side. No token is outputted to the left side in y. To the right side, nothing is outputted.

(2) When the cell has been occupied with the data tokens:

The control token T is outputted from R to the left side, and it is shown that data are available. Concurrently with this, the data token that has been maintained is outputted from y. To the right side, the control token F is fed from C′ and pop of the data is demanded.

Complying with the above, the control token T is returned from R′ from the right side, if data are available. Concurrently with this, the data token is fed from y′. As a result, the control token T is maintained on the upper side, and the data token fed from the right is maintained on the lower side.

If no data are available in the right side, the control token F is fed from R′and nothing is fed from y′. As a result, the control token F is maintained on the upper side of this cell and there remains a state of no token being left.

As is seen from the above, the part in which the data are maintained exclusively causes the action as a matter of reality. None of the other parts causes any actions. By employing DSTA, let the procedure DSTACK be composed as shown in Fig. 4.1(b).
The actions of DSTACK are shown in Table 4.1, and is illustrated in fig,4.2.

TABLE 4.1 Actions of procedure DSTACK.

	Input			Output	
	Z	C	x	R	y
No Action	F	any	none	F	none
Pushdown d_0	T	T	d_0	F	F
Popup	T	F	none	†	‡

† indicates F, if the stack has been empty. If not, it indicates T.
‡ indicates the data that has been placed on the head ofthe stack. If no data has been placed there, nothing is outputted.
'any' indicates that each of T or F is acceptable.

On the other hand, let what has changed T-gate, F-gate, and merge concerning the data tokens in the lower part of DSTA into the ones all for the control tokens be CSTA. Furthermore by using this CSTA, let CSTACK be composed as shown in Fig. 4.3. The actions of CSTACK are shown in Table 4.2.

Table 4.2 Actions of procedure CSTACK.

	Input			Output	
	Z	C	x	R	y
No Action	F	any	any	F	F
Pushdown c_0	T	T	c_0	F	F
Popup	T	F	any	†	‡

† indicates F, if the stack has been empty. If not, it indicates T.

‡ indicates the true-false value that has been placed on the head of the stack.
If no token has been placed there, nothing is outputted.
'any' indicates that each of T or F is acceptable.

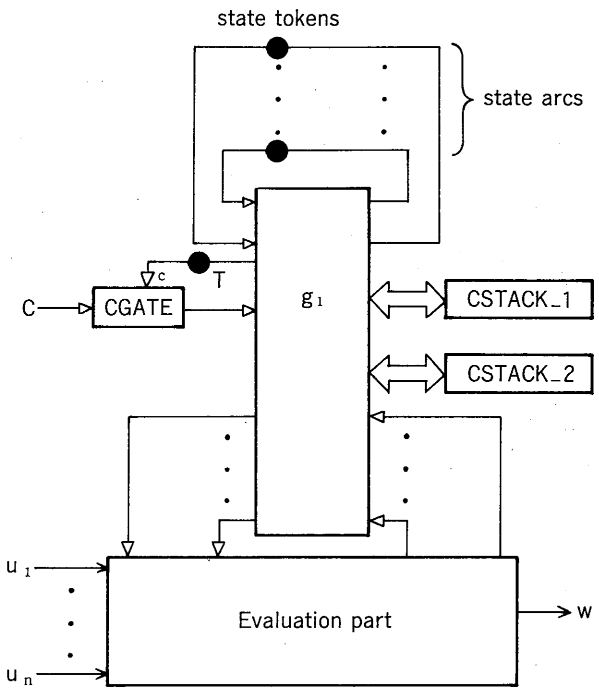
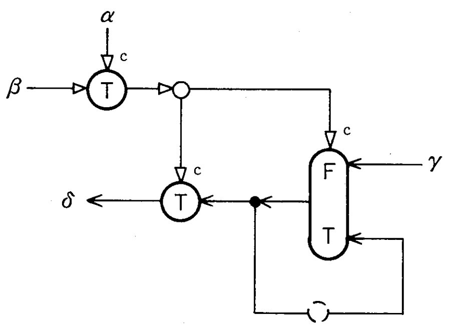
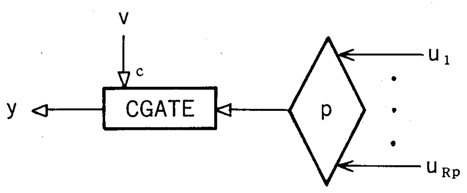


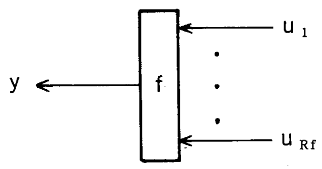
Fig. 4.4 Procedure COMPUTE_1 (w:,u_1,...u_n,C)



(a) Latch



(b) predicate evaluator



(c) function evaluator

Fig. 4.5 some elements included in the evaluating part.

As is shown in[6], an arbitrary finite-state machine can be composed by linking the output arc of the boolean graph to the input arc in a shape of a loop, Furthermore, an arbitrary Turing machine can be simulated by linking 2 pieces of CSTACK. By linking an evaluation part to the Turing machine, the procedure COMPUTE_1 allowing a computation to be evaluated can be composed as shown in Fig. 4.4. Here, g_1 is the boolean graph taking part of the Turing machine. By taking in at first the token sequence $T^i F$ expressing the natural number i and later by imitating the Turing machine $\mathcal{T}$, the procedure composes the i-th computation $\mathcal{T}$(i) on a piece of CSTACK. In the computation in question, let the individual expressions or predicates be expressed in accordance with the post-fix description method and furthermore let proper sectioning symbols be placed among them.

With the finite control part, let $\mathcal{T}$(i) be evaluated by contolling the evaluation part after $\mathcal{T}$(i) is obtained.

The evaluation part includes a latch(Fig. 4.5(a)) corresponding to the individual input variables. This always maintains the input data in a position of circle drawn in broken line, and outputs its copies in responce to necessity. On the other hand, the evaluation part includes a predicate evaluator for the individual predicate symbols(Fig. 4.5(b)), and also includes a function evaluator for the individual function symbols (Fig.4.5(c)).

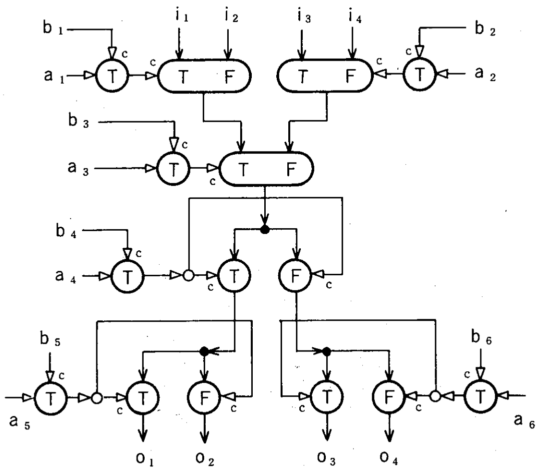

i_1 i_2 i_3 i_4
b_1 b_2 b_3 b_4 b_5 b_6
a_1 a_2 a_3 a_4 a_5 a_6
T F
c
o_1 o_2 o_3 o_4

Fig. 4.6 An example of 4 to 4 transfer network

Furthermore the part includes a transfer network in order to allow the data tokens to be transmitted among these individual elements. The transfer network is the one to transmit a data token from an arbitrary point to another point in accordance with the command from the finite control part. The important property of the transfer network is that once it is demanded to transfer a data token from one point to another, and no data token is inputted to the input point, following demand to transfer a data token is obstructed. There is a point where any data token transferd by the network should go through. The network is composed of two tree structures connected to each other at their root points. One is to collect a data token from a leaf to the root, and is made of data-merges. The other is provide a data token from the root to a leaf, and is made of pairs of data T-gate and F-gate. An example of a transfer network with four inputs and four outputs is shown in Fig.4.6. By giving a proper tokens through a_i and b_i, the finite controller can transfer a data token from either point of $i_1,\ldots,i_4$ to either point of $o_1,\ldots,o_4$.

The finite control part evaluates the computation as shown below using such an evaluation part. First of all, let the command to house the input data in the individual latches be issued. Secondly, let evaluation be made by successively reading the codes of $\mathcal{T}$(i) from CSTACK_1.

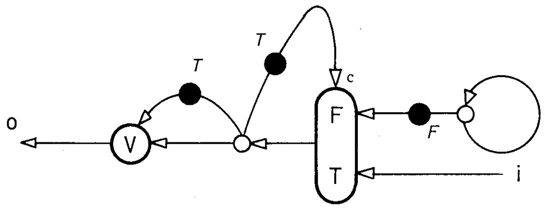
When the code of $x_i \in \mathbf{X}$ is read, let the token be taken out from the latch of x_i to be pushed to DSTACK.

When the code of $f_i \in \mathbf{F}$ is read, let Rf_i pieces of the data placed on the head of DSTACK be taken out and sent to the f_i evaluator. Let the evaluation result thus obtained be pushed to DSTACK. When the code of $p_i \in P$ is read, let Rp_i pieces of the data be taken out and sent to the p_i evaluator. Let the result thus obtained be taken in. If the result taken in is T then let the evaluation be continued. When the final expression is evaluated, let the value be outputted to W then cease the action. When a predicate having the value of F is found in the course, cease the action. If a predicate becomes undefined, the finite control part is obstructed because the control token from the predicate is not inputted. If a function becomes undefined, the data token from the function is not inputed to the inputpoint of the transfer network, and then the result of the evaluation of the final expression could not be transfered to the output.

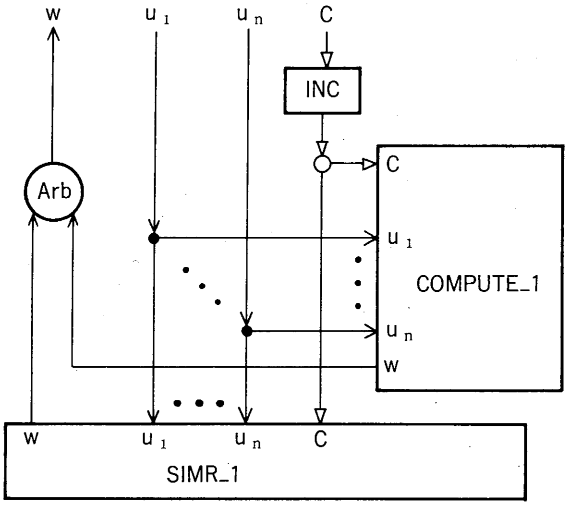
As is comprehended from the above, for both case of $\mu=1$ or ∞, the procedure COMPUTE_1 is a procedure to evaluate the computation $\mathcal{T}$(i) by taking in the token sequence expressing the natural number i. When the procedure possesses a value, T and the value are outputted. When a predicate of the falsity value is found in the course, F is outputted.

Theorem 4.1 (EF$\leqq$ADF(μ)) : In both cases that $\mu=1$ and $\mu=\infty$, for an arbitrary schema *S* of *EF*, there exists an equivalent schema *S'* of *ADF*(μ).
[Proof] By using the procedure COMPUTE_1, let a recursive procedure SIMR_1(Fig.4.7(b)) be composed. The procedure INC(Fig4.7(a)) is the one adding 1 to the value of i expressed by the token sequence. That is to say, the procedure is the one to allow the evaluation starting with the 1st computation to be made succesively. If there is at least one computation which gives a result, the data token is transfered through arbiters. By composing the program SIMULATE_1 using this procedure as Fig.4.7(c), any schema $S \in EF$ is evidently simulated.
Q.E.D.

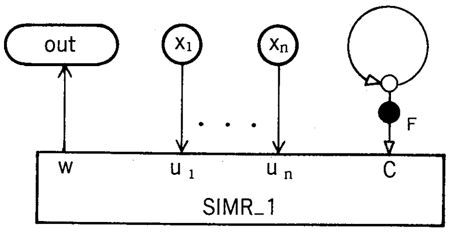
Theorem 4.2(*ADF*(∞)$\leqq$*EF*):For any schema *S* of the class *ADF*(∞), there is an equivalent schema *S'* of *EF*.
[Proof]The action of S can be simulated by a nondeterministic Turing machine $\mathcal{N}$ in a symbolic manner. The actions of $\mathcal{N}$ which stoppes can be enumelated



(a) Procedure INC(o:i)



(b) Procedure SIMR_1(w:u_1,...,u_n,C)



(c) Program SIMULATE_1(out:x_1,...,x_n)

Fig. 4.7 Program SIMULATE_1

and the corresponding computation can be constructed. It is possible to construct a deterministic Turing machine $\mathcal{T}$ which takes a natural number i as input and outputs the i-th computation. S' can be constructed using $\mathcal{T}$. Q.E.D.

Corollary 4.3($ADF(\mu) \equiv EF$)

[Proof]It is evident from the two theorems.

Q.E.D.

5 $DF_\pi(\infty) \geqq DF_{arb}(\infty)$

The π-gate is introduced instead of the arbiter. However, these two devices have the common property of the timing dependency, which means that the value of the output token depends on which token on two input arcs is inputted faster. The π-gate seemes to be more useful than the arbiter since the π-gate is controllable by the control token. As a matter of fact, we show the relation

$DF_\pi(\infty) \geqq DF_{arb}(\infty)$.

Theorem 5.1($DF_\pi(\infty) \geqq DF_{arb}(\infty)$):For each schema S of $DF_{arb}(\infty)$, there exists an equivalent schema S' of $DF_\pi(\infty)$.

[Proof]The arbiter of S can be simulated by the statements of S' as depicted in the figure 5.1. In the figure, the part labeled with g_2 is a Boolean graph

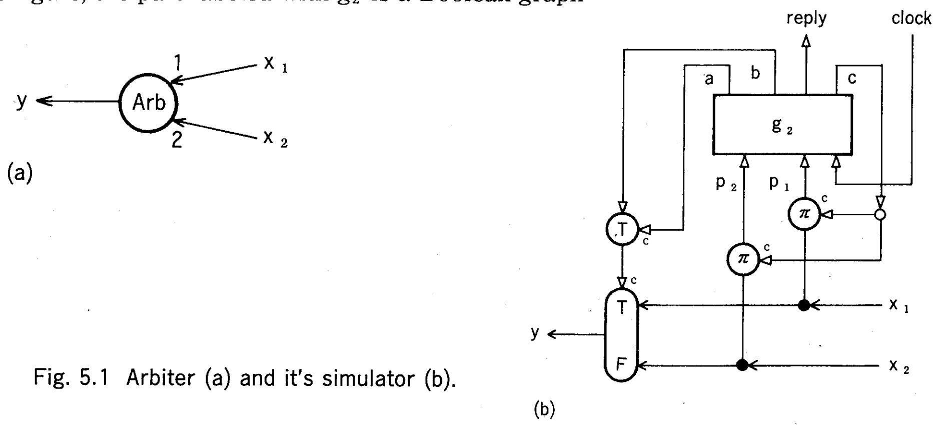


Fig. 5.1 Arbiter (a) and it's simulator (b).

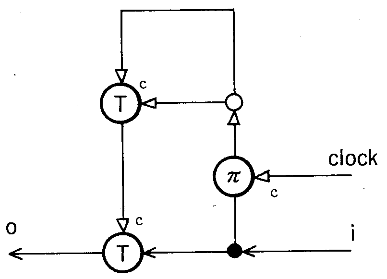


Fig. 5.2 Output unit attached to the statements.

which realize a function described in the table 5.1. The part is always sencing the tokens on input arcs. When no token is there, the part instructs no action to the merge by sending F thru a. When there exist a token on x_1, the part instructs the merge to transfer the token to y by sending T to a and b. When there exists a token on x_2 and no token on x_1, the merge transfers the token to y.

Strictly speaking, the simulator takes more time than the original arbiter. Therefore, it may be possible that the delay cansed by the simulator affect the actions on the S' and then yield the differnt result from S. To avoid this possibilty, we should make the actions of other statements as slow as the simulator. For this purpose, the output unit is provided to each statement. We prepare the central clock unit which prvoides clock tokens to each output unit and arbiter simulator. The clock unit outputs the clock token only when all of the reply tokens have returned from each arbiter simulators. The value of the clock token is T disregarding the value of reply tokens. The output unit transfers the input token to the output arc, when the clock unit is provided. However, if there is no token on the input arc when the clock token from the central clock token is provided, the unit does nothing. Thus each statement cannot act faster than the arbiter simulator. Decider and operator can act faster than the original in a relative meaning. This makes no problem, however, because the schema yields the same result for any time function consistent with the interpretaion.

Q.E.D.

Table 5.1 The function realized by g_2.

p_1	p_2	a	b	c
F	F	F	F	F
F	T	T	F	F
T	F	T	T	F
T	T	T	T	F

6 Conclusions

To answer the problem whether both of arbiter and recursive procedure are necessary for *ADF* to become equivalent to *EF*, it is necessary to investigate the expressive power of the classes which have single device. Further, it is also important to distinguish between the case that an arc can hold arbitrary number of tokens and the case that at most one token can exist on an arc. Until now, the expressive powers of the classes $ADF(1)$, $DF_{\pi}(\infty)$, $DF_{arb}(1)$, $RDF(\infty)$, $RDF(1)$, $DF(\infty)$, and $DF(1)$ are investigated in comparison with the classes EF, EF^{d} and P. However, the expressive powers of the class $ADF(\infty)$ and $DF_{arb}(\infty)$ has not been investigated. In this paper, we have shown the class $ADF(\infty)$ also is equivalent to EF. As to the class $DF_{arb}(\infty)$, it was difficult to establish the proper inclusion relation between the class and $DF_{arb}(1)$. Instead of the class, we have introduced the class $DF_{\pi}(\infty)$. The class $DF_{\pi}(\infty)$ occupies a important position in the whole inclusion relations. Though the class $DF_{arb}(\infty)$ takes rather a miner role in the whole inclusion relations, we are intersted in where the class shoud be located. In this paper, we have also shown the relation $DF_{\pi}(\infty) \geqq DF_{arb}(\infty)$. It is not clear, however, whether the relation is proper or not. It is evident the relation $DF_{arb}(\infty) \geqq DF(\infty)$ is satisfied. Therefore, if the inclusion relation between $DF_{arb}(\infty)$ and $DF_{arb}(1)$is established, some proper inclusion relations be induced. Thus the problem, however, is an open problem for the present.

References

[1] Dennis,J.B. and Misunas,D.P.,"A preliminary architecture for a basic data-flow processor", comp.Struc. Group Memo.102, Project MAC, MIT(Dec 1977).

[2] Kusakabe,S.,Takahashi,E.,Taniguchi,R. and Amamiya,M.,"A dataflowbased massively parallel programming language, V,and its implementation on a commercially available parallel machine.", Trans. IPS. Japan. 36,7,pp.1529-1541(1995).

[3] Dennis,J.B.,"First version of data flow procedure language", Lecture Notes in computer Science,5, Springer-Verlag, (1974).

[4] Constable,R.L. and Gries,D.,"On classes of program schemata",SIAM Journal on computing,1,1,pp.66-118(1972).

[5] Strong,H.R.,"High level languages of maximum power",PROC. IEEE Conf. on Switching and Automata Theory, pp.1-4(1971).

[6] Jaffe,J.M.,"The Equivalence of r.e.program schemes and data flow schemes", JCSS,21, pp.92-109(1980).

[7] Matsubara,Y.and Noguchi,S.,"Expressive power of data flow schemes on partial Interpretations",Trans. of IECE Japan, J67-D,4. pp.496-503(1984)

[8] Matsubara,Y. and Noguchi,S.,"Data flow schemata of maximum expressive power", Trans. of IECE Japan, J67-D,12, pp.1411-1418(1984).
[9] Matsubara,Y.,"Ability of the class RDF(∞) of Dataflow Schemata with Open Recursive Procedure",submitting.
[10] Matsubara,Y.,"Ability of the class of recursive dataflow schemata with arcs holding at most one token",submitting.
[11] Matsubara,Y.,"Ability of the class of dataflow schemata with timing dependency", submitting.