数理解析研究所講究録
第 436 巻 1981 年 174-197

ABSTRACT PROGRAM MODEL AND FUNCTIONAL MAPPING

M.Ohba, Y.Tanitsu, N.Takimoto, H.Kadota

Product Assurance Laboratory, IBM Japan, Ltd.

ABSTRACT

Programs consist of algorithms and data structures. Extending this well-known basic concept, algorithms can be regarded as consisting of algoritms for control-flow and algorithms for data-flow. On the basis of the separation of control and data flow, the functor model is proposed to describe the interaction among co-operating concurrent modules within the given programming system, and the functor kernel is also proposed to specify the internal control flow of each functor model as a pushdown-automaton.

1. INTRODUCTION

Generally, programs are regarded as consisting of algorithms and data structures. Algorithms can furthermore be regarded as consisting of control-flow for defining control mechanisms and data-flow for defining operations on data. It is therefore important to visualize the interaction among co-operating concurrent modules and their internal control mechanisms, in order to specify and verify the given programming system.

Recently, some interesting works have been done in the related areas. H.D.Mills has proposed PDL/Ada as a program design language on a basis of the Turing machine and the function semantics. The IBM Communication Systems Architecture group has developed another design language and its processor on a basis of the finite state automaton in order to describe the network architecture. The La Gaude IBM group has also developed another finite state automaton based language and its interpreter.

The advantage of the Turing machine based model such as PDL/Ada is a precise description, while the advantage of the finite state model is a concise description. On the other hand, the PDL and other Turing machine based models are to much complicated to overview the asserted algorithms at the higher level of abstraction. Hence, it is sometimes hard to understand what is asserted in the described model. The disadvantage of the finite state models is, as pointed out by D.S.Scott, most of complicated programs cannot be modeled.

It is important for solving this semantic gap problem to allow a proper balance between rigorous formulation and conceptual simplicity. R.Kowalski suggests to split the given algorithm into the control component and the logic component to approach the balancing problem. In his approach, the control component is treated as an environment for the logic component, like as the operating system for the problem program. The control component determines how the logic components works to perform the functions efficiently.

There is another approach suggested by R.D.Tennent. In his approach, the body of the program, the algorithm, is regarded as a construct of control rules and functions for operations on data.

The functor model which is discussed in this paper is influenced by his approach for balancing preciseness and compactness of the description. For allowing a clear, provable, and comprehensive architectural specification for the target programming system, the functor model is specified by defining the interaction among co-operating modules called functors and the internal control flow of each functor. The interaction among functors is defined as a directed graph called the functor net which is similar to the Petri-net. The internal control of each functor is defined as a non-deterministic pushdown automaton.

The functor model is a practical method to visualize the control rules among co-operating modules within a programming system and within each module which is regarded as consisting of task synchronization controls : e.g., WAIT, POST, etc.; internal sequence controls: e.g., DO, IF-THEN-ELSE, etc.; and further detail data manipulation functions: e.g., MOVE, READ, WRITE, arithmetic operations, etc.

2. GENERAL CONCEPT

To construct a model that represents the data-processing algorithm, it is reasonable to split the algorithm into the decision layer algorithm and the data-manipulation layer algorithm. In this layer scheme, the decision layer controls the flow of processing data, and the data-manipulation layer performs the detail function for processing data. In this paper, the decision layer algorithm is called the kernel, and the data-manipulation layer algorithm is called the data-flow.

From the architectural point of view, the kernel describes how the functions described in the data-flow is controlled. The data-flow describes how the abstract data is processed within the abstract processor. Abstract data consist of data structures and data relations refered by the data-flow symbolically. Data entities are data objects which are actually processed by the data-flow.(Fig.1) Under the multi-processing(and/or multi-programming) environment, the kernel is furthermore regarded as consisting of the process(task) synchronization rules and the process internal sequence control rules. The

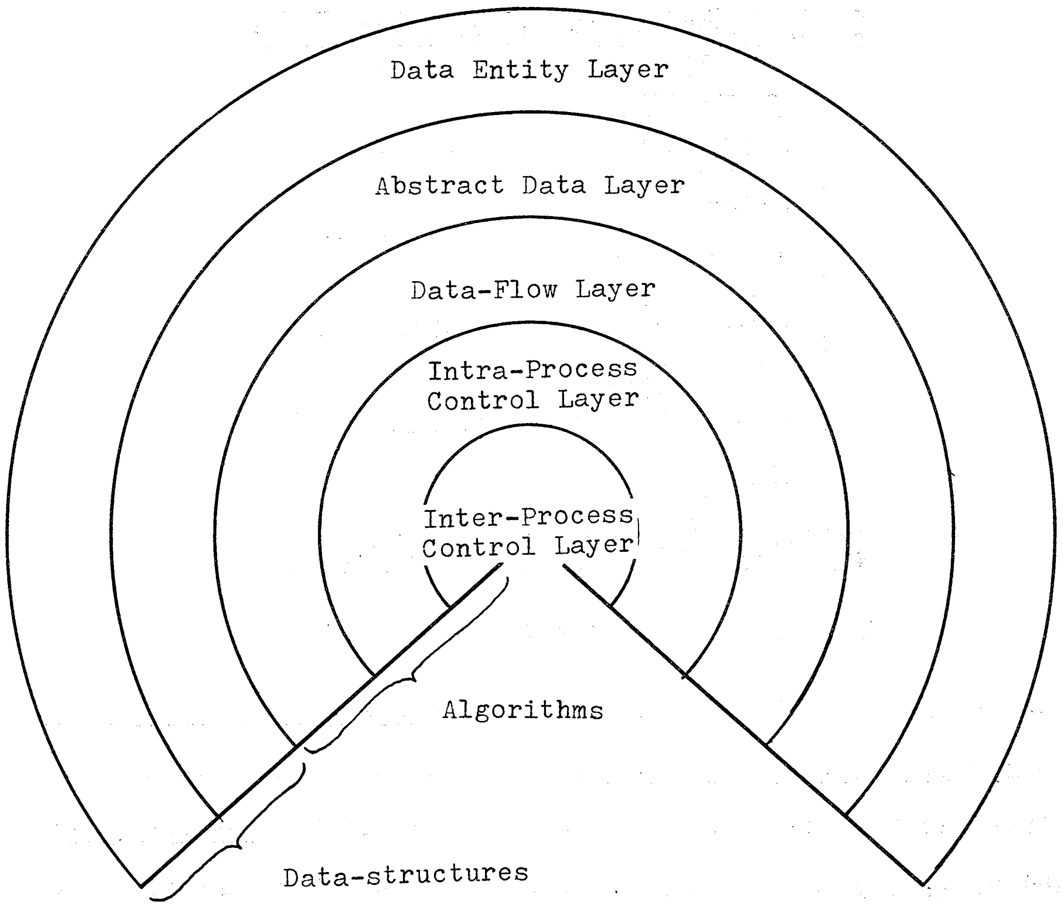


Fig.1 Layer structure

abstract processor is a kind of the actor which simulates the behavior of the target data-processing system under the abstract processor architecture. The abstract processor architecture is a multi-processor architecture that consists of a finite number of abstract processors. Under the abstract processor architecture, each abstract processor is connected to the inter-processor signal channel and the inter-processor data channel. The inter-processor signal channel transfers symbolic signals from one abstract processor to another. The inter-processor data channel transfers data entities from one abstract processor to another.(Fig.2)

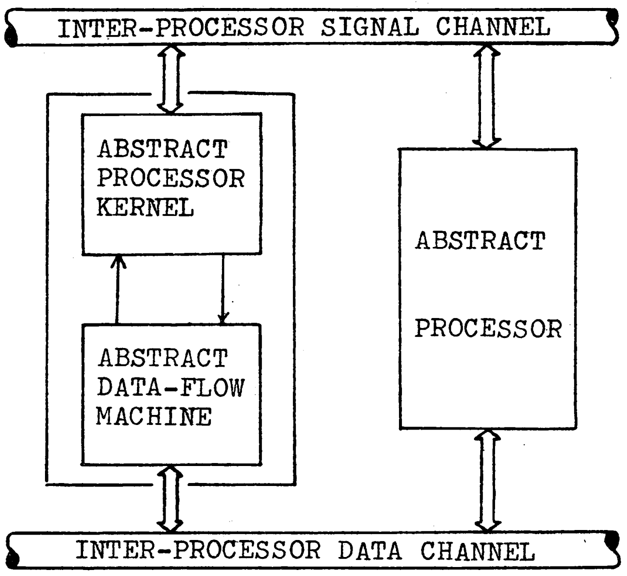

INTER-PROCESSOR SIGNAL CHANNEL
ABSTRACT PROCESSOR KERNEL
ABSTRACT DATA-FLOW MACHINE
ABSTRACT PROCESSOR
INTER-PROCESSOR DATA CHANNEL

Fig.2 Abstract processor architecture

An abstract processor consists of an abstarct processor kernel and an abstract data-flow machine. For processing data, an abstract processor has two phases in the execution cycle: the control phase and the process phase. During the control phase, the abstract processor determines what function should be performed in the succeeding process phase on the basis of the results of the previous process phase.

During the process phase, the abstract processor processes data entities according to the order from the previous control phase. After data have been processed, the abstract data-flow machine informs the abstract processor kernel of the result of data-processing to determine the next process during the subsequent control phase.

The abstract processor kernel can be modelled as a PLA, which produces out-bound signals as either the reply to other abstract processors or orders to specify what function of the abstract data-flow machine should be activated during the subsequent process phase, based on the in-bound signal as either the request from another abstract processor or the response from the abstract data-flow machine as a result of the process. The abstract processor kernel is therefore equivalent to the specifications of the target data-processing machine at the higher-level of abstraction.

3. FUNCTOR MODEL

The functor is an abstract program model executed by the abstract processor system. Corresponding to each layer of the abstract processor architecture, the abstract program model is also hierarchically structured as consisting of the description of control-flow and the description of data-flow. The kernel scheme describes higher-level functions of the abstract program by defining control-flow operationally within the abstract program and its interface denotationally. The data-flow scheme describes detail functions of the abstract program by specifying procedures for operations on data.(Fig.3)

The functor is a pd-automaton model of the kernel scheme. For

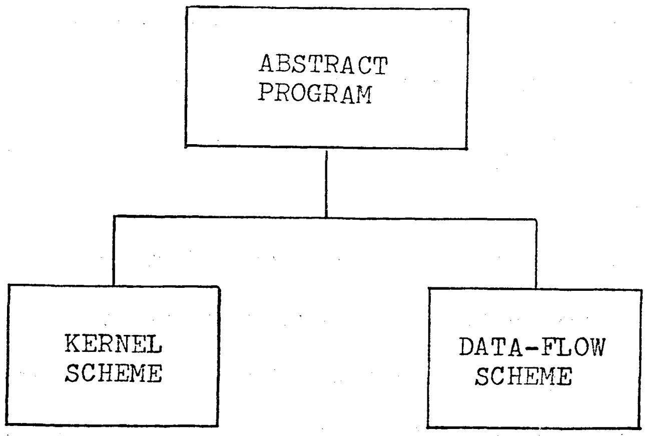


Fig.3 Abstract program model

specifying the higher-level model of an abstract program system which consists of two or more concurrent programs, abstract programs are regarded as consisting of a set of independedt pd-automata which co-operate interactively. As the correspondence of the abstract program to the inter-processor signal channel in the abstract processor architecture, the gate is introduced for syncronization and communication among co-operating functors. Therefore, a given abstract program system can be modelled as a construct of functors and gates.

There are six different types of functors and three different types of gates for modelling co-operating abstract programs. Six different types of functors are the initial functor, the terminal functor, the transit functor, the selective functor, the merging functor, and the continual functor. Three different types of gates are the F-gate, the J-gate, and the E-gate. The initial functor is a starting point of abstract program control-flow where control resides at the time of system initialization. The terminal functor is a vanishing point of abstract program control-flow where control is absorbed for program termination. The transit functor is a relaying point of abstract program control-flow through which control passes. The selective functor is a disjunction point of abstract program control-flow from which control branches off. The merging functor is a conjunction point of abstract program control-flow where control-flow branched at the selective functor is merged togather. The continual functor is a non-functional functor which indicates the originator of the POST operation for synchronization between co-operating abstract programs. Except for the continual functor, other five types of functors perform their indivisual functions. The initial functor and the merging functor are introduced to describe initialization processes, while the terminal functor and the selective functor are introduced to describe termination processes.

The F-gate is a fork node of concurrent abstract program control-flow where two or more processes(sub-tasks) start concurrently, and introduced to describe the ATTACH and the POST operation. The J-gate is a join node of concurrent abstract program control-flow where two or more concurrent processes(sub-tasks) are synchronized and merged, and introduced to describe the WAIT and the multiple WAIT operation. The E-gate is another type of the join node which describe the countable WAIT operation.(Fig.4)

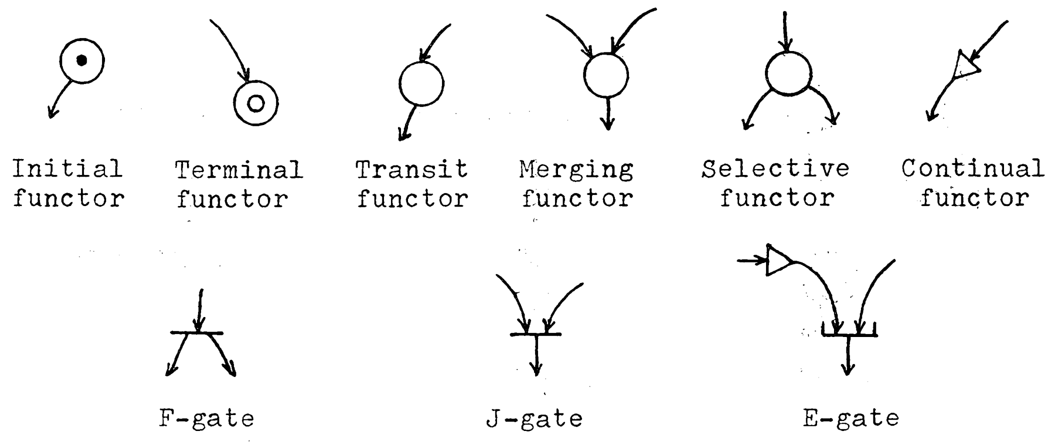
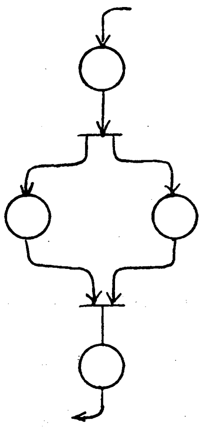
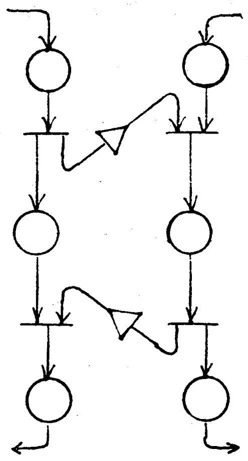


Fig.4 Functors and gates



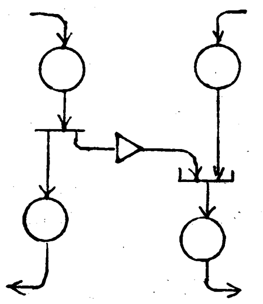
```
Main-task          Sub-task X          Sub-task Y
    .                  .                   .
    .                  .                   .
ATTACH X               .                   .
ATTACH Y               .                   .
    .              POST ECBM           POST ECBM
    .                  .                   .
WAITM ECBM             .                   .
    .                  .                   .
    .              RETURN              RETURN
DETACH X
DETACH Y
    .
    .
```

(a) Initiation and termination of sub-tasks



```
Sub-task X          Sub-task Y
    .                   .
    .                   .
    .                   .
POST ECBY           WAIT ECBY
    .                   .
    .                   .
WAIT ECBX           POST ECBX
    .                   .
    .                   .
```

(b) Synchronization between sub-tasks



```
Sub-task X          Sub-task Y
    .                   .
    .                   .
    .                   .
POSTC ECBY          WAITC ECBY
    .                   .
    .                   .
```

(c) Message passing between sub-tasks

Fig.5 Example of the functor representation

Example (a) of Fig.5 represents a typical example of multi-task programming: the main-task initializes two sub-tasks, X and Y, by means of issuing ATTACH macro, and waits until these sub-tasks complete their processes. Example (b) represents a typical example of the inter-process synchronization: the sub-task X posts the sub-task Y and waits until the sub-task Y posts it. Example (c) represents a typical example of the message-passing between sub-tasks: the sub-task X prepares data for the sub-task Y and passes it to the sub-task Y by means of issuing POSTC macro, and the sub-task Y receives data by means of issuing WAITC macro. Definitions of POSTC and WAITC macro are as below:

```
definition of POSTC                 definition of WAITC
POSTC ecb;                          WAITC ecb;
  begin                               begin
    COUNT = COUNT + 1                   if COUNT = 0 then WAIT ecb
    if COUNT = 1 then POST ecb          COUNT = COUNT - 1
  end;                                end;
```

As a result of the above extension of the Petri-net, initialization and termination of the abstract program can be clearly specified(Fig.6), and the inter-process control such as WAIT-POST control between sub-tasks is also explicitly specified. Although the graph representation of the Petri-net is powerful for modelling concurrent programs, inter-task control-flow is not visible enough when the graph model becomes a large scale one. Because the Petri-net does not distinguish inter-process control-flow from intra-process control-flow.(Fig.7(a)) The Petri-net therefore requires semantic interpretation of the graph to distinguish inter-process control-flow from intra-process control-flow. On the other hand, the functor net classifies these two control-flow

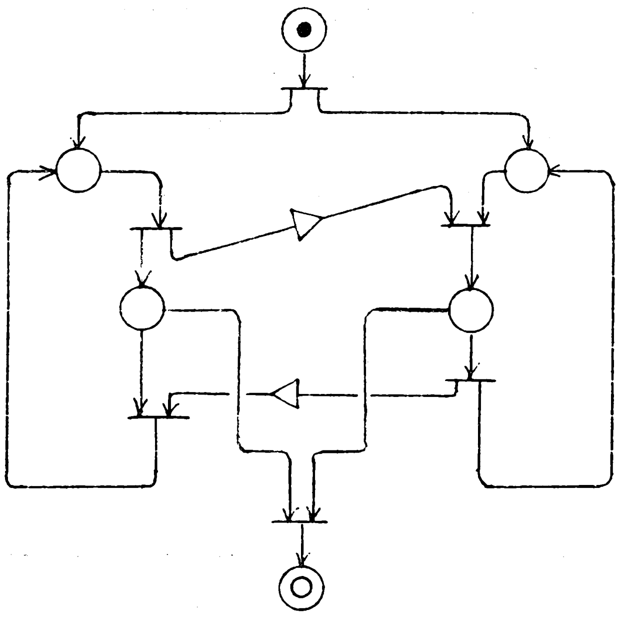
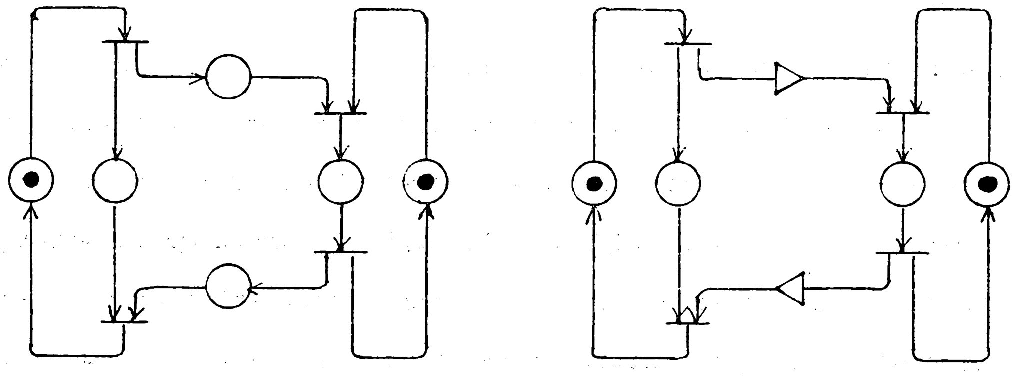


Fig.6 The functor model with initialization/termination



(a) Petri-net model (b) Functor model

Fig.7 The functor model and the Petri-net

as different classes using different notations.

4. EXAMPLE OF FUNCTOR MODEL

Fig.8 shows a typical example of multi-programming structure models in the online application. The example program consists of the main-task A, the receive sub-task X, the process sub-task Y, and the send sub-task Z. The rough structure of the program is shown in Fig.9.

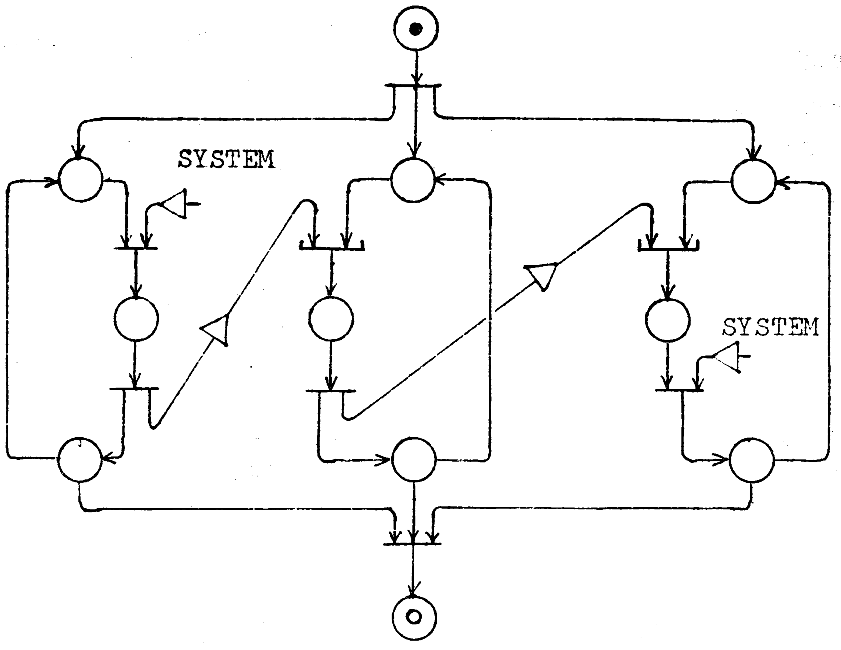


Fig.8 Example model of online application program

In the example program, the receive sub-task X receives data from a terminal, checks received data whether or not shut-down is requested, decrements the local variable COUNT and passes the shut-down request to the process sub-task Y if shut-down is requested; or passes the received data to the sub-task Y if shut-down is not requested, posts the sub-task Y by means of issuing POSTC macro, and repeats the above procedure until the local variable COUNT becomes zero. The process sub-task Y waits for data from the receive sub-task X, is posted when the message comes from

```
MAIN-TASK A
  BEGIN
    .
    .
    ATTACH  X
    ATTACH  Y
    ATTACH  Z
    .
    .
    WAITM   ECBA
    .
    .
    DETACH  X
    DETACH  Y
    DETACH  Z
    .
    .
  END
```

```
SUB-TASK X
  BEGIN
    COUNT=N
    .
    .
    DO UNTIL COUNT=0
      RECEIVE
      WAIT  ECBX
      IF REQUEST=SHUT-DOWN
        THEN COUNT=COUNT-1
             pass shut-down-req
        ELSE pass data
      ENDIF
      POSTC ECBY
    ENDDO
    .
    .
    POST  ECBA
  END
```

```
SUB-TASK Y
  BEGIN
    COUNT=N
    .
    .
    DO UNTIL COUNT=0
      WAITC ECBY
      get message from X
      IF message=shut-down-req
        THEN COUNT=COUNT-1
             pass message
        ELSE process data
             pass data
      ENDIF
      POSTC ECBZ
    ENDDO
    .
    .
    POST  ECBA
  END
```

```
SUB-TASK Z
  BEGIN
    COUNT=N
    .
    .
    DO UNTIL COUNT=0
      WAITC ECBZ
      get message from Y
      IF message=shut-down-req
        THEN COUNT=COUNT-1
             SEND SHUT-DOWN
        ELSE SEND data
      ENDIF
      WAIT  ECBZZ
    ENDDO
    .
    .
    POST  ECBA
  END
```

Fig.9 Example program structure

the sub-task X, checks the message whether or not shut-down is requested decrements the local variable COUNT and passes the shut-down request to the send sub-task Z if shut-down is requested; or processes data for sending back to the terminal and passes processed data to the send sub-task Z if shut-down is not requested, posts the sub-task Z by means of issuing POSTC macro, and repeats the above until the local variable COUNT becomes zero. The send sub-task Z waits for the message from the sub-task Y, is posted when the message comes from the sub-task Y, checks the message and performs either decrements the local variable COUNT and sends the shut-down command to the terminal if the message is a shut-down request; or sends processed data to the terminal if the message is a send-data request, waits for completion by the access method routine of the operating system, and repeats the above until the local variable COUNT becomes zero. When the local variable COUNT becomes zero which means no active sessions exist, all sub-tasks post the main-task for termination.

What is described in the functor graph model shown in Fig.8, can be interpreted as follows:

```
the main-process always forks sub-processes X, Y, and Z;
  sub-processes concurrently operate and sometimes repeat:
    the sub-process X always passes the request to the sub-process Y;
    the sub-process Y always passes the request to the sub-process Z;
  sub-processes sometimes join into the main-process;
the main-process always ends.
```

5. FUNCTIONAL MAPPING

As described previously, the functor is an abstract program of which control structure can be described as a pd-automaton, and consists of a control-flow component called the functor kernel and data-flow components called procedures. Structure of the functor kernel can be defined as consisting of its external input (request) from other functors, its internal input (response) from procedures, its internal state, its internal control variables, its internal output (order) to procedures, and its external output (reply) to other functors. From the external (denotational) point of view, the functor produces the reply for the request from another functor. From the operational point of view, the functor kernel controls the sequence of operations within the functor for producing the output by means of issuing orders to data-flow procedures depending upon its internal state, its internal control variables, the response from the procedure, and the request received from another functor. The internal state of the functor always changes depending upon progress of operations within the functor, while internal control variables occasionally change depending upon the requirement of memorizing new values for these variables. Typical examples of the internal control variables are program switches, do-loop counters, and do-loop termination control variable such that the parameter N of the flow-chart in Fig.10 is a representative.

The functor kernel is characerized as equations below:

$$\begin{cases} S' = \mathfrak{F}_{\gamma}(\mathfrak{i}(S);\ \mathfrak{C}(S);\ S) \\ \mathfrak{O}(S') = \mathfrak{F}_{\varphi}(\mathfrak{i}(S);\ \mathfrak{C}(S);\ S) \\ \mathfrak{C}(S') = \mathfrak{F}_{\varkappa}(\mathfrak{i}(S);\ \mathfrak{C}(S);\ S), \end{cases}$$

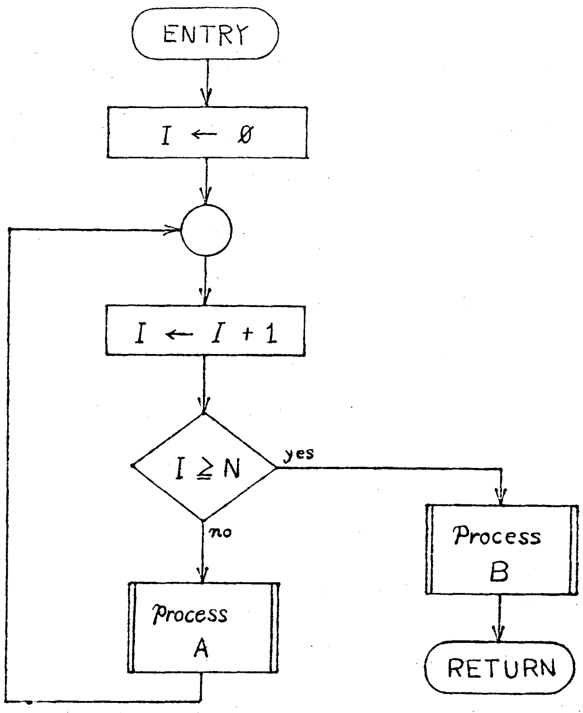


Fig.10 Example of DO-loop Termination Control Variable

where S is the internal state of the functor; S' is the next internal state of the functor when state-transition from the current state S completed; $\mathbb{C}(S)$ is the internal control variable vector at the state S ; $\mathring{\mathfrak{i}}(S)$ and $\mathbb{O}(S')$ are defined as follows,

$$\mathring{\mathfrak{i}}(S) = \begin{bmatrix} \mathring{\mathfrak{i}}^d(S) \\ \mathring{\mathfrak{i}}^c(S) \end{bmatrix},$$

$$\mathbb{O}(S') = \begin{bmatrix} \mathbb{O}^d(S') \\ \mathbb{O}^c(S') \end{bmatrix},$$

where $\mathring{\mathfrak{i}}^d(S)$ is the external input vector at the state S , $\mathring{\mathfrak{i}}^c(S)$ is the internal input vector (response from procedures) at the state S , $\mathbb{O}^d(S')$ is the external output vector (reply to other functors) at the state S', and $\mathbb{O}^c(S')$ is the internal output vector (order to procedures) at the state S'.

Let the domain $\mathcal{S}_{\mathfrak{F}}$ be a set of states of the given functor $\mathfrak{F}$, such that $\mathcal{S}_{\mathfrak{F}}$ satisfies: for any state s of $\mathfrak{F}$: $\{ \mathcal{S}_{\mathfrak{F}} \cup \{s\}\} \subseteq \mathcal{S}_{\mathfrak{F}}$. Let the domain $\mathcal{C}_{\mathfrak{F}}$ be a Cartesian product of domains $\mathcal{C}_{\mathfrak{F}}^{(1)}$, $\mathcal{C}_{\mathfrak{F}}^{(2)}$, ..., and $\mathcal{C}_{\mathfrak{F}}^{(m)}$, where $\mathcal{C}_{\mathfrak{F}}^{(i)}$ is a set of all possible values of the internal control variable $c^{(i)}$ of the given functor $\mathfrak{F}$, such that $\mathcal{C}_{\mathfrak{F}}^{(i)}$ satisfies: for any value $c_j^{(i)}$ of $c^{(i)}$, $\{ \mathcal{C}_{\mathfrak{F}}^{(i)} \cup \{c_j^{(i)}\}\} \subseteq \mathcal{C}_{\mathfrak{F}}^{(i)}$. Let the domain $\mathcal{I}_{\mathfrak{F}}$ be a Cartesian product of domains $\mathcal{I}_{\mathfrak{F}}^{d(1)}$, $\mathcal{I}_{\mathfrak{F}}^{d(2)}$, ..., $\mathcal{I}_{\mathfrak{F}}^{d(k)}$, $\mathcal{I}_{\mathfrak{F}}^{c(1)}$, $\mathcal{I}_{\mathfrak{F}}^{c(2)}$, ..., and $\mathcal{I}_{\mathfrak{F}}^{c(l)}$, where $\mathcal{I}_{\mathfrak{F}}^{d(i)}$ is a set of all possible values of the external input variable $i^{d(i)}$ and $\mathcal{I}_{\mathfrak{F}}^{c(i)}$ is a set of all possible values of the internal input (response from procedures) variable $i^{c(i)}$ of the given functor $\mathfrak{F}$, such that $\mathcal{I}_{\mathfrak{F}}^{d(i)}$ and $\mathcal{I}_{\mathfrak{F}}^{c(i)}$ satisfy: for any value $i_j^{d(i)}$ of $i^{d(i)}$ and $i_j^{c(i)}$ of $i^{c(i)}$, $\{ \mathcal{I}_{\mathfrak{F}}^{d(i)} \cup \{ i_j^{d(i)}\}\} \subseteq \mathcal{I}_{\mathfrak{F}}^{d(i)}$, and $\{ \mathcal{I}_{\mathfrak{F}}^{c(i)} \cup \{i_j^{c(i)}\}\} \subseteq \mathcal{I}_{\mathfrak{F}}^{c(i)}$. Let the domain $\mathcal{O}_{\mathfrak{F}}$ be a Cartesian product of domains $\mathcal{O}_{\mathfrak{F}}^{d(1)}$, $\mathcal{O}_{\mathfrak{F}}^{d(2)}$, ..., $\mathcal{O}_{\mathfrak{F}}^{d(k')}$, $\mathcal{O}_{\mathfrak{F}}^{c(1)}$, $\mathcal{O}_{\mathfrak{F}}^{c(2)}$, ..., and $\mathcal{O}_{\mathfrak{F}}^{c(l')}$, where $\mathcal{O}_{\mathfrak{F}}^{d(i)}$ is a set of all possible values of the external output variavle $o^{d(i)}$ and $\mathcal{O}_{\mathfrak{F}}^{c(i)}$ is a set of all possible values of the internal output (order to procedures) variable $o^{c(i)}$ of the given functor $\mathfrak{F}$, such that $\mathcal{O}_{\mathfrak{F}}^{d(i)}$ and $\mathcal{O}_{\mathfrak{F}}^{c(i)}$ satisfy: for any value $o_j^{d(i)}$ of $o^{d(i)}$ and $o_j^{c(i)}$ of $o^{c(i)}$, $\{ \mathcal{O}_{\mathfrak{F}}^{d(i)} \cup \{o_j^{d(i)}\}\} \subseteq \mathcal{O}_{\mathfrak{F}}^{d(i)}$, and $\{ \mathcal{O}_{\mathfrak{F}}^{c(i)} \cup \{o_j^{c(i)}\}\} \subseteq \mathcal{O}_{\mathfrak{F}}^{c(i)}$. The functor kernel is modelled and structured as consisting of mappings φ, γ , and ψ , as shown in Fig.11. The functional mapping model shown

$$\begin{tikzcd}
 & \mathcal{O}_{\mathfrak{F}} \\
\mathcal{I}_{\mathfrak{F}} \times \mathcal{S}_{\mathfrak{F}} \times \mathcal{C}_{\mathfrak{F}} \arrow[ru, "\varphi"] \arrow[r, "\psi"] \arrow[rd, "\gamma"'] & \mathcal{C}_{\mathfrak{F}} \\
 & \mathcal{S}_{\mathfrak{F}}
\end{tikzcd}$$

Fig.11 The functional mapping

in Fig.11 implies that the proper domain $\mathcal{N}_{\mathfrak{F}}$ of finite integers can be determined iff domains $\mathcal{I}_{\mathfrak{F}}$ and $\mathcal{C}_{\mathfrak{F}}$ are finite. Introducing the domain $\mathcal{N}_{\mathfrak{F}}$, the model illustrated in Fig.11 is transformed as Fig.12.

$$\begin{tikzcd} & & \mathfrak{O}_{\mathfrak{F}} \\ \mathfrak{I}_{\mathfrak{F}} \times \mathfrak{S}_{\mathfrak{F}} \times \mathfrak{C}_{\mathfrak{F}} \arrow[r, "\sigma"] & \mathfrak{N}_{\mathfrak{F}} \arrow[ur, "\mu"] \arrow[r, "\lambda"] \arrow[dr, "\kappa"'] & \mathfrak{C}_{\mathfrak{F}} \\ & & \mathfrak{S}_{\mathfrak{F}} \end{tikzcd}$$

Fig.12 The functional mapping with domain

Semaintics of the functional mapping model illustrated in Fig.12, is as follows. The mapping $\sigma : \mathfrak{S}_{\mathfrak{F}} \times \mathfrak{C}_{\mathfrak{F}} \times \mathfrak{I}_{\mathfrak{F}} \rightarrow \mathfrak{N}_{\mathfrak{F}}$, analyzes the environment of operations and identifies a condition of the current operation. The mapping $\mu : \mathfrak{N}_{\mathfrak{F}} \rightarrow \mathfrak{O}_{\mathfrak{F}}$, determines the output of the current operation on the basis of the identified condition by the mapping μ. The mapping $\lambda : \mathfrak{N}_{\mathfrak{F}} \rightarrow \mathfrak{C}_{\mathfrak{F}}$, determines new values of internal control variables as a result of the current operation. The mapping $\kappa : \mathfrak{N}_{\mathfrak{F}} \rightarrow \mathfrak{S}_{\mathfrak{F}}$, determines the next state of the functor $\mathfrak{F}$ as a result of the current operation. The new environment of operation, that is induced by new values of control variables and the new state, determines the range of possible actions for the further operations of the functor. Therefore, the functional mapping specifies the high-level semantic interpretation of the given functor $\mathfrak{F}$.

6. ALGEBRAIC MODEL OF FUNCTIONAL MAPPING

In case where input variables, internal state variable, internal control variables, and output variables are given as Boolean variable vectors, the algebraic representation of functor kernel functions $\mathfrak{F}_\sigma$, $\mathfrak{F}_\mu$, $\mathfrak{F}_\lambda$, and $\mathfrak{F}_\kappa$ is discussed.

For manipulation between Boolean vector and matrix, following two algebraic operations $\dot\wedge$ and $\dot\vee$ are introduced.

DEFINITION: the operation $\dot{\wedge}$ is defined over the Boolean matrix $\mathbb{A}$ and the Boolean vector $\mathbb{b}$, such that;

$$\mathbb{A} \dot{\wedge} \mathbb{b} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix} \dot{\wedge} \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}$$

$$= \begin{bmatrix} (a_{11} \blacktriangledown b_1) \wedge (a_{12} \blacktriangledown b_2) \wedge \cdots \wedge (a_{1m} \blacktriangledown b_m) \\ (a_{21} \blacktriangledown b_1) \wedge (a_{22} \blacktriangledown b_2) \wedge \cdots \wedge (a_{2m} \blacktriangledown b_m) \\ \vdots \qquad\qquad \vdots \qquad\qquad \vdots \\ (a_{m1} \blacktriangledown b_1) \wedge (a_{m2} \blacktriangledown b_2) \wedge \cdots \wedge (a_{mm} \blacktriangledown b_m) \end{bmatrix},$$

The result of the operation is an Boolean vector which consists of m Boolean variables.

DEFINITION: the operation $\blacktriangledown$ is defined over two Boolean variables p and q, such that the operation satisfies the truth table below;

p	q	$p \blacktriangledown q$
T	T	T
T	F	F
F	T	F
F	F	T

Table 1 Truth table of operation

DEFINITION: the operation $\dot{\vee}$ is defined over the Boolean matrix $\mathbb{A}$ and the Boolean vector $\mathbb{b}$, such that;

$$\mathbb{A} \dot{\vee} \mathbb{b} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1m} \\ a_{21} & a_{22} & \cdots & a_{2m} \\ \vdots & \vdots & & \vdots \\ a_{l1} & a_{l2} & \cdots & a_{lm} \end{bmatrix} \dot{\vee} \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{bmatrix}$$

$$= \begin{bmatrix} (a_{11} \wedge b_1) \vee (a_{12} \wedge b_2) \vee \cdots \vee (a_{1m} \wedge b_m) \\ (a_{21} \wedge b_1) \vee (a_{22} \wedge b_2) \vee \cdots \vee (a_{2m} \wedge b_m) \\ \vdots \qquad\qquad \vdots \qquad\qquad \vdots \\ (a_{l1} \wedge b_1) \vee (a_{l2} \wedge b_2) \vee \cdots \vee (a_{lm} \wedge b_m) \end{bmatrix},$$

The result of the operation is an Boolean vector which consists of l Boolean variables.

Applying operations $\underset{\cdot}{\wedge}$ and $\dot{\vee}$, the function $\mathfrak{F}_{\varphi}$ can be expressed as the equation below:

$$\begin{aligned}
\mathbb{O}(\mathbb{S}') &= \mathfrak{F}_{\varphi}(\ \mathring{i}(\mathbb{S});\ \mathbb{C}(\mathbb{S});\ \mathbb{S}\ ) \\
&= \mathfrak{F}_{\mu}(\ \mathfrak{F}_{\sigma}(\ \mathring{i}(\mathbb{S});\ \mathbb{C}(\mathbb{S});\ \mathbb{S}\ )) \\
&= \mathbb{M}\dot{\vee}(\ \Sigma\ \underset{\cdot}{\wedge}\ \begin{bmatrix} \mathring{i}(\mathbb{S}) \\ \mathbb{C}(\mathbb{S}) \\ \mathbb{S} \end{bmatrix}\ ) \quad .
\end{aligned}$$

Similarly, the function $\mathfrak{F}_{\gamma}$ and $\mathfrak{F}_{\chi}$ can be expressed as:

$$\begin{aligned}
\mathbb{S}' &= \mathfrak{F}_{\gamma}(\ \mathring{i}(\mathbb{S});\ \mathbb{C}(\mathbb{S});\ \mathbb{S}\ ) \\
&= \mathfrak{F}_{\kappa}(\ \mathfrak{F}_{\sigma}(\ \mathring{i}(\mathbb{S});\ \mathbb{C}(\mathbb{S});\ \mathbb{S}\ )) \\
&= \mathbb{K}\ \dot{\vee}(\ \Sigma\ \underset{\cdot}{\wedge}\ \begin{bmatrix} \mathring{i}(\mathbb{S}) \\ \mathbb{C}(\mathbb{S}) \\ \mathbb{S} \end{bmatrix}\ ) \quad ,
\end{aligned}$$

$$\begin{aligned}
\mathbb{C}(\mathbb{S}') &= \mathfrak{F}_{\chi}(\ \mathring{i}(\mathbb{S});\ \mathbb{C}(\mathbb{S});\ \mathbb{S}\ ) \\
&= \mathfrak{F}_{\lambda}(\ \mathfrak{F}_{\sigma}(\ \mathring{i}(\mathbb{S});\ \mathbb{C}(\mathbb{S});\ \mathbb{S}\ )) \\
&= \Lambda\ \dot{\vee}(\ \Sigma\ \underset{\cdot}{\wedge}\ \begin{bmatrix} \mathring{i}(\mathbb{S}) \\ \mathbb{C}(\mathbb{S}) \\ \mathbb{S} \end{bmatrix}\ ) \quad .
\end{aligned}$$

Therefore, the functor kernel model is given by the equation:

$$\begin{bmatrix} \mathbb{O}(\mathbb{S}') \\ \mathbb{C}(\mathbb{S}') \\ \mathbb{S}' \end{bmatrix} = \begin{bmatrix} \mathbb{M} \\ \Lambda \\ \mathbb{K} \end{bmatrix} \dot{\vee}(\ \Sigma\ \underset{\cdot}{\wedge}\ \begin{bmatrix} \mathring{i}(\mathbb{S}) \\ \mathbb{C}(\mathbb{S}) \\ \mathbb{S} \end{bmatrix}\ ) \quad ,$$

where, $\mathbb{K}$, Λ , and $\mathbb{M}$ are Boolean matrices which are defined by the specifications of the functor $\mathfrak{F}$.

Let the vector $\mathfrak{x}$ be an Boolean variable vector, such that:

$$\mathfrak{x} = \begin{bmatrix} \mathring{i}(\mathbb{S}) \\ \mathbb{C}(\mathbb{S}) \\ \mathbb{S} \end{bmatrix} \quad .$$

Let the matrix $\mathbb{A}$ be an Boolean constant matrix, such that:

$$\mathbb{A} = \begin{bmatrix} \mathbb{M} \\ \Lambda \\ \mathbb{K} \end{bmatrix} \quad .$$

And, let the vector $\mathfrak{y}$ be an Boolean variable vector, such that:

$$\mathbb{Y} = \begin{bmatrix} \mathbb{D}(\mathbb{S}') \\ \mathbb{C}(\mathbb{S}') \\ \mathbb{S}' \end{bmatrix}$$

Consequently, the functor kernel model is given by the equation:

$$\mathbb{Y} = \mathbb{A} \dot{\vee} (\mathbb{\Sigma} \underset{\cdot}{\wedge} \mathbb{X}) \; .$$

Semantics of the above equation is operationally defined as Fig.13. The language used for the description of semantics is PL/I. The procedure named KERNEL describes semantics of the above equation. It is assumed in the procedure that the vector XX corresponding to the vector $\mathbb{X}$, the vector YY corresponding to the vector $\mathbb{Y}$, the array SS corresponding to the matrix $\mathbb{\Sigma}$, and the array AA corresponding to the matrix $\mathbb{A}$ are defined externally and passed from the calling procedure, and dimensions of arrays and vectors are also passed from the calling procedure. The procedure SNOC notifies that the should not occure condition, for example undefined input detected, occures.

The algebraic model described above is very equivalent to PLA's (Programmable Logic Arrays) of the hardware logic component. Same as PLA's sometimes free the hardware logics from the architecture, the basic structure of the functional mapping is independent from the detail functions of the model to be described. The matrix $\mathbb{\Sigma}$ corresponds to AND-array of the PLA, and matrices $\mathbb{K}$, $\mathbb{\Lambda}$, and $\mathbb{M}$ correspond to OR-array.

Furthermore, the model can be regarded as an extension of the decision table. The decision table logic is inherently static: it does not depend upon states and internal control variables. The functional mapping model can describe dynamic logics by means of utilizing state and internal control variables.

```
BEGIN;
  DECLARE L,M,N BINARY FIXED;        /* DIMENSION OF ARRAYS */
  DECLARE II(128) BIT;               /* INPUT VECTOR        */
  DECLARE SS(768,128) BIT;           /* SIGMA MATRIX        */
  DECLARE MM(768,384) BIT;           /* MU    MATRIX        */
  DECLARE OO(384) BIT;               /* OUTPUT VECTOR       */
    :
    :
    L=128;
    M=768;
    N=384;
  CALL KERNEL(II,SS,MM,OO);
  /***********************/
  /* PROCEDURE : KERNEL */
  /***********************/
KERNEL: PROCEDURE (XX,SS,AA,YY);
  /* DECLARATION PART */
  DECLARE XX(*),SS(*,*),AA(*,*),YY(*) BIT;
  DECLARE SW1,SW2 BIT;               /* CONTROL VARIABLES   */
  DECLARE I,J BINARY FIXED;          /* DO LOOP COUNTER     */
  /* PROCEDURE BODY */
  SW1='0'B;                          /* SET SW1 OFF         */
  DO I=1 TO M WHILE(SW1='0'B);       /* APPLY SIGMA MATRIX  */
    SW2='0'B;                        /* SET SW2 OFF         */
    DO J=1 TO L WHILE(SW2='0'B);     /* CHECK ALL ELEMENT   */
      IF SS(I,J) -= XX(J)            /*  SS(I,*)=XX(*)      */
        THEN SW2='1'B;               /*  IF TRUE THEN       */
    END;                             /*    SW2 STAYS OFF    */
    IF SW2='0'B                      /* IF SS(I,*)=XX(*)    */
      THEN SW1='1'B;                 /*  THEN SW1 = OFF     */
  END;
  IF SW1='1'B                        /* IF SS(I,*)=XX(*)    */
    THEN                             /* THEN                */
      DO J=1 TO N;                   /* APPLY MU MATRIX     */
        IF AA(I,J)='1'B              /*   YY(*):=AA(I,*)    */
          THEN YY(J)='1'B;           /*   FOR ALL ELEMENT   */
          ELSE YY(J)='0'B;           /*   OF MATRIX MU      */
      END;
    ELSE                             /* ELSE                */
      CALL SNOC;                     /*  SHOULD NOT OCCURE  */
                                     /*  CONDITION OCCURES  */
  END;                               /* RETURN TO THE CALLER*/
    :
    :
END;                                 /* END OF BLOCK        */
```

Fig.13 Operational definition of the functional mapping model

7. CONCLUSION

The functor model described in this paper is constructed on the basis of the Petri-net, the Actor theory, and the decision table. The general concept of the abstract program structurization and the abstract processor architecture is influenced by the Actor theory. The graphic representation method of concurrent abstract programs is influenced by the Petri-net. The algebraic model of the abstract program internals is influenced by the decision table and the PLA. The algebraic representation of the functional mapping is another interpretation of the state vector semantics for specifying the abstract program functions.

These method of modelling programs in the actual use environment at different levels of abstraction, are hierarchically structured from the top level to down levels: at the top level, the Actor theory is a powerful tool for defining how system components interactively work within the target system; at the second level, the Petri-net is a powerful tool for defining how modules within the system co-operate and the concurrency of the system is mechanized; at the third level, the state vector approach is powerful for defining how each module works under the concurrency control mechanism defined at the second level; at the fourth level, the data-flow language may be a powerful tool for defining how data are processed within a module under the sequence control defined at the third level. At this point of time, the functor model and the algebraic model of the functional mapping are therefore effective and practical tools for engineering the software.

The further research effort is currently planned for the following areas: the algebraic representation of concurrency control mechanisms, the formal language for the concurrent program specification, and the validation method of the concurrency algorithm described in the formal language.

Finally, the authors deeply thank Mr. A. O'hara, who works for System Communications Assurance of the IBM Kingston laboratory, for his technical comments and enthusiastic discussions with the authors.

REFERENCES:

1) Yonezawa A; "Specifying Software Systems with High Internal Concurrency Based on Actor Formalism", Journal of Info. Processing, Vol.2 No.4, 1980.

2) Lautenbach K; "Use of Petri Nets for Proving Correctness of Concurrent Process Systems", Proc. of Info. Processing 74, 1974.

3) Priese L; "A Note on Asynchronous Cellar Automata", Journal of Computer & Systems Sciences 17, 1978.

4) Jotwani N et al.; "Top-down Design in the Context of Parallel Programs", Journal of Information & Control 40, 1979.

5) Conry S et al.; "On Functional Equivalences in a Model for Parallel Computation", Journal of Information & Control 41, 1979.

6) Mekly L et al.; "Software Design Representation Using Abstract Process Networks", IEEE Trans. on Software Engineering, Vol.SE-6, No.5, 1980.

7) Leung K et al.; "Logical System Design using PLAs and Petri Nets", Proc. of Info. Processing 77, 1977.

8) Ohba M et al.; "Architecture Kernel: Higher Level Program Specification", Proc. of COMPSAC80, 1980.