TR-0792

A Petri-Net-Based Programming Environment and its Design Methodology for Cooperating Discrete Event Systems

by
N. Uchihira, M. Arami & S. Honiden (Toshiba)

August, 1992

ICOT
Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
(03)3456-3191～5
Telex ICOT J32964
Minato-ku Tokyo 108 Japan

Institute for New Generation Computer Technology

A Petri-Net-Based Programming Environment and its Design Methodology for Cooperating Discrete Event Systems

Naoshi UCHIHIRA, Mikako ARAMI, Shinichi HONIDEN

Systems & Software Engineering Laboratory,
TOSHIBA Corporation,
Kawasaki 210, JAPAN

Keywords:
High Level Petri Net, Cooperating Discrete Event System, Concurrent Program, Design Methodology, Program Synthesis, Program Verification, Temporal Logic, Programming Environment.

SUMMARY: This paper describes MENDELS ZONE, a Petri-net-based concurrent programming environment, which is especially suitable for cooperating discrete event systems. MENDELS ZONE adopts MENDEL net, which is a type of high level (hierarchical colored) Petri net. One of the characteristics of the MENDEL nets is a process-oriented hierarchy like CCS, which is different from the subnet-oriented hierarchy in the Jensen's hierarchical colored Petri net. In a process-oriented hierarchy, a hierarchical unit is a process, which is more natural for cooperating and decentralized discrete event control systems. This paper also proposes a design methodology for MENDEL nets. Although many Petri net tools have been proposed, most tools support only drawing, simulation, and analysis of Petri nets; few tools support the design methodology for Petri nets. While Petri nets are good final design documents easy to understand, analyzable, and executable, it is often difficult to write Petri nets directly in an earlier design phase when the system structure is obscure. A proposed design methodology makes a designer to construct MENDEL nets systematically using causality matrices and temporal logic. Furthermore, constructed MENDEL nets can be automatically compiled into a concurrent programming language and executed on a parallel computer.

1. INTRODUCTION

There is an increasing demand for programmers who can design concurrent programs, as the use of practical parallel and distributed computer systems gradually spread in the industry. Since it is not easy for ordinary programmers to produce correct and efficient concurrent programs, several kinds of CASE (Computer-Aided Software Engineering) tools are necessary. Some examples include tools for verification, debugging, performance evaluation, and synthesis of correct and efficient programs. MENDELS ZONE (1)(2)(3) is a CASE environment which has been under development since 1986. It facilitates the difficult task of concurrent programming for cooperating discrete event systems. A cooperating discrete event system (CDES) means a discrete event system that consists of several cooperating subsystems and coordinators, such as distributed manufacturing systems and communication network systems. We distinguish CDESs from sequential discrete event systems which are easier to design and for which practical CASE tools have already been developed (e.g., SFC tools (4)). The difficulties in designing CDES are mainly caused by the fact that its global behaviors become complex with the combinatorial state explosion and can not be fully grasped and expected by the designer. Petri net is one of the most hopeful tools, whose graphical representation of concurrency and various analysis methods can remedy the difficulties. MENDELS ZONE provides computer-aided concurrent program construction tool which uses high level Petri nets. Our approach has two appealing features described below:

First, a new high level Petri net is proposed. Up to now, several high-level Petri nets and tools have been proposed, which include Colored Petri Nets (CPN) and its tool (DESIGN/CPN) (5), Predicate/Transition Nets (6), and Algebraic Petri Nets (7). CPN (i.e., DESIGN/CPN) introduces hierarchy constructs into nets to enable a large-scale system description (8). These hierarchy constructs are subnet-oriented; that is, a part of the net (subnet) is regarded as a hierarchical unit and reduced to one node. This subnet-oriented hierarchy is inadequate to represent the compositional structure of concurrent processes. For example, DESIGN/CPN cannot directly specify the process composition of Process Theory (e.g., CCS (9)). The CCS-like compositional structure is necessary to represent interactions of cooperating processes in CDES. Therefore, we propose a process-oriented hierarchy, which allows the processes, process composition, and synchronous/asynchronous communication between them to be explicitly represented. MENDEL net is a high-level Petri net adopting the process-oriented hierarchy. MENDELS ZONE supports a designer in the compositional construction of CDES using MENDEL nets. Furthermore, constructed MENDEL nets can be compiled into a concurrent programming language KL1 and be executed on a parallel computer Multi-PSI.

Second, a Petri-net-based design methodology for CDES is proposed. Most tools that have been proposed so far support only the drawing (graphic editor), simulation, and analysis (reachability and invariant analysis) of Petri nets (10)(11). While Petri nets are good final design documents that are easy to

understand, analyzable, and executable, it is often difficult to write Petri nets directly in an earlier design phase when the system structure is obscure. In the design methodology we propose, a designer can construct MENDEL nets systematically through all design phases using wide-spectrum causality matrices and temporal logic. MENDELS ZONE provides not only a MENDEL net editor, verification and adjustment tools, and a compiler to the concurrent programming language but also provides a computer-aided design methodology.

The remainder of the paper is organized as follows. Section 2 defines a MENDEL net, a type of high-level Petri net. The design methodology for MENDEL nets is described in Section 3. A brief overview of MENDELS ZONE is given; we explain its supporting tools for the design methodology. Section 5 describes a MENDEL net design example for a lift system, followed by a conclusion and related works in Section 6.

2. MENDEL NET

A MENDEL net is a high level Petri net in addition to ordinary Petri nets that has the following properties:
- three types of places (state, slot, and port),
- logic program description of transition conditions and actions,
- priority of transitions,
- process-oriented net hierarchy,
- two types of communication mechanisms (synchronous and asynchronous) between processes, and
- Array representation.

The above properties not only make MENDEL nets powerful enough to describe most cooperating discrete event systems but also make it possible to retract automatically a skeleton of the MENDEL nets to be used for a subsequent net analysis phase (verification and adjustment). MENDEL nets are designed to handle detailed descriptions as well as skeleton-level analysis. The following subsections will explain the above properties in detail and also mention concurrent program code generation from MENDEL nets.

2.1 Place

The places and transitions of ordinary Petri nets are very general and can have a wide variety of meanings. From the viewpoint of concurrent programs, places are classified into three types (state elements, slots, and ports) (see **Fig. 1**):

– state element: a place which represents the local state of a system. This type of place has at most one token (i.e. it is safe). If the place has one token, it means that the system stays in the local state (called "the state element is ON"). Otherwise, it means that the system does not stay in the local state (called "the state element is OFF"). The state elements are graphically represented by circles.

– slot: a place that represents data and database on a static storage. This type of place has constantly one token. The slots are graphically represented by horizontal bars similar to the Data Flow Diagram of Structured Analysis.

– port: a place that represents an infinite buffer necessary for modeling data flow and asynchronous communication. The ports are graphically represented by ellipses.

This classification produces informative structures utilized in Petri net design, analysis, understanding, and code generation.

2.2 Transition

In MENDEL net, the transition of Petri net is called a **method**. A method is graphically represented by a rectangle. The method's firing conditions and actions are described in detail by the concurrent logic programming language KL1 (12). Therefore, a MENDEL net is a kind of colored Petri net, where the colors of tokens are represented in logic programming terms (atom, integer, logical variable, and list), and the conditions and actions are described with guards and bodies of KL1 clauses, respectively. The textual form of a method follows:

method(*method_name*, *exchange_term*,
input_state_list, *output_state_list*,
input_slot_list, *output_slot_list*,
input_port_list, *output_port_list*, *priority*) :-
KL1_guard | *KL1_body* ;

The exchange_term is used for synchronous communication which is described later. The slot and port are textually described in the form: *slot_name*(*KL1_term*) and *port_name*(*KL1_term*). There are two levels of method priority: normal and express. A method will be briefly explained by the following example (**Fig.2**). (Refer to the MENDEL language manual (13) for the detailed syntax and semantics of the textual form of a MENDEL net).

Method Example:

```
method(move,_,
      [ready], [busy],
      [x(N1), y(M1)], [x(N2), y(M2)],
      [type(T)], [ack(A)], normal) :-
      N1>0, M1>0, T=job(ID) | N2:=N1+1, M2:=M1+1, A=ok(ID) ;
```

In this example, the method "move" is enabled if **(C1)** the state element "ready" is ON, **(C2)** there exists at least one token in the port "type" whose color can be unified with T, **(C3)** the KL1 guard (N1>0 and M1>0 and T=job(ID)) is true where terms N1 and M1 are read from the slots "x" and "y", and **(C4)** there are no other enabled methods with an "express" priority. When the method "move" is fired, **(A1)** the KL1 body (N2:=N1+1, M2:=M1+1, A=ok(ID)) is evaluated, **(A2)** the state element "ready" becomes OFF and the state element "busy" becomes ON, **(A3)** the evaluated terms N2 and M2 are written in the slots "x" and "y", and **(A4)** the token is removed from the port "type" and a token whose color is ok(ID) is pushed into the port "ack".

2.3 Hierarchy

The hierarchical unit of MENDEL nets is a process. A process may consist of several subprocesses. The interaction between hierarchical units is modeled as synchronous and asynchronous communication between processes. The process interface is a set of external ports and external methods. A process can push/pop tokens to/from external ports of subprocesses. Since ports are infinite buffers, this interaction realizes asynchronous communication. On the other hand, a process can synchronize its own methods with the external methods of subprocesses (i.e., fire these methods simultaneously only if they are all enabled). Since data exchange is available using the exchange_term, this interaction realizes synchronous communication.

Figure 3(a) illustrates a simple MENDEL net example including a process-oriented hierarchy. The external ports and external methods are represented by double ellipses and double rectangles, respectively. From the viewpoint of the parent process, a subprocess is graphically represented by a large circle (don't confuse it with a state element), and the external ports and methods of the subprocess are represented by bars (P-plug) and dots (M-plug) attached to the large circle. A bar represents a postbox for asynchronous communication, and a dot represents an outlet for synchronous communication. Asynchronous communication is represented by an arrow between a method and a P-plug, and synchronous communication is represented by a dotted line between a method and an M-plug.

We emphasize that this process-oriented hierarchy can directly specify a compositional structure of well-researched concurrent process theories, such as CCS, ACP, CSP, and LOTOS. For example, the CCS expression ($P = Q[med/out] \mid Q[med/in]$, $Q = in.\tau.out.Q$), where a process P is defined as composition of two processes Q, can be directly represented by a MENDEL net as shown in **Fig. 4(b)**. It cannot be done by a subnet-oriented hierarchy which supprts no synchronous communication. G-LOTOS [14], a graphical representation with a process-oriented hierarchy of LOTOS, is similar to MENDEL net. However, G-LOTOS supports no asynchronous communication.

2.4 Array Representation

Occasionally, a CDES contains several subprocesses having the same structure. For example, a lift system may have several request buttons corresponding to floors. It is tedious to individually write all button processes. To overcome this problem, colored Petri nets represent these subprocesses as separate colored tokens on a single net structure. However, this approach is not suitable for the process-oriented hierarchy, because a process is a hierarchical unit and tokens should not be a process. Therefore, a MENDEL net provides an alternative: an array to represent several subprocesses with the same structure, in the same manner as a CSP-based concurrent programming language Occam [15]. Each of arrayed processes can be explicitly treated as a separate computing unit that can run on a separate CPU. **Figure 4** shows a graphical

array representation of N identical processes, whose textual form follows (Refer to MENDEL manual [13]):

```
process_{I}(m_{I})(in_{I})(out_{I})*I:{1..N}
```

2.5 Concurrent Program Generation

In Most Petri net tools used for the system design, target programs have to be manually coded from designed Petri nets. Our aim is to automatically generate target programs directly from MENDEL nets where generated programs run efficiently in the real environment. Design/CPN provides the automatic generation of SML (Standard ML) codes [5]. However, SML is not a **concurrent** programming language. MENDEL nets can be compiled into concurrent programs (KL1 codes) and executed efficiently on a parallel computer, which will be also described in the explanation of MENDELS ZONE (Section 4).

3. DESIGN METHODOLOGY

This section proposes a new design methodology for MENDEL nets using causality matrices and temporal logic.

3.1 Design Principle

As mentioned in Section 1, Petri nets are often inadequate in an early design phase when some parts of the system structure may be obscure. In particular, the structures of a coordinator (task scheduler, synchronizer) in a CDES are often obscure at first, while the structures of individual local DESs (local tasks) can be well defined. Since a Petri net is a formal language and does not permit vagueness, it is difficult to design systems from beginning to end using only Petri net.

This paper proposes a design methodology for CDES utilizing MENDEL net and additional complementary formalisms (causality matrix and temporal logic). The causality matrix allows vagueness (abstract level description), so a designer can stepwise refine the causality matrix from an abstract level to a concrete level. Furthermore, draft Petri nets which are synthesized from the causality matrix are verified and adjusted (tuned up) to satisfy given temporal logic constraints. By doing these stepwise refinements, the designer can construct a correct MENDEL net systematically.

3.2 Causality Matrix

The causality matrix, $C=[c_{ij}]$, is a type of an extended incidence matrix of a Petri net (**Fig. 5**), where each entry, c_{ij}, represents a causality relation between an i-th operatior (i.e., method) and a j-th operand (i.e., state element, slot, port, or external method (M-plug)). One of unique features of the causality matrix is its wide-spectrum property that the causality relation of each entry can range from an abstract level to a concrete level (we call this a wide-spectrum causality

relation). **Table 1** shows wide-spectrum causality relations. For example, c_{11}="+" of **Fig. 5** is an abstract level relation representing only existence of relation between the operator *opr1* and the operand *opn1*; the details of the relation are not given at this level. On the other hand, c_{23}="pop:X" is a concrete level relation in which the operator *opr2* pops a token from the operand *opd3* and unifies the color of the token with X (i.e., X is assigned the color of the token). A designer stepwise refines operators, operands, and causality relations from an abstract level to a concrete level, which is regarded as a design process. The wide-spectrum causality matrix has the following advantages compared with graphical Petri net representations.
(1) The binary relation in the matrix is more adequate for system element analysis in the earlier design phase. It is much easier for a designer to decide what is a local abstract relation between every two elements than to directly draw a rigid Petri net.
(2) A spread-sheet editor for the matrix can provide powerful editing abilities. Copying, eliminating, decomposing, browsing, focusing, and checking can be implemented more easily than by a graphical one. Furthermore, graphical representations are difficult to perceive without good topological arrangement. Current automatic topological arrangements in Petri net tools are not yet good enough.
(3) The wide-spectrum property makes backtracking easier. Backtracking is inevitable in a design process. However, the wide-spectrum property enables the designer to manipulate only common and consistent design documents (causality matrices) throughout the design process, which minimizes the modification efforts in backtracking.

A concrete causality matrix can be transformed straightforward into a process of MENDEL nets. In other words, a causality matrix is a matrix-based representation of MENDEL nets which allows vagueness and informality. **Figure 6** shows an example of straightforward transformation from a matrix to a process of MENDEL nets. Remark that each matrix corresponds to one process and does not have a hierarchical structure in itself, which should be represented by MENDEL nets.

3.3 Temporal Logic

Linear time Propositional Temporal Logic (LPTL) is a propositional logic ($\neg$: NOT, $\wedge$: AND, $\vee$: OR, and $\supset$: IMPLY) that is extended with several temporal operators (**F** f: "f will be eventually true" and **G** f: "f is always true"). To combine a Petri net with LPTL, it is assumed that an atomic proposition corresponds to method firing (i.e., an atomic proposition **m** is true if and only if a method **m** fires). We can declaratively specify the following properties of Petri nets by LPTL:
(1) mutual exclusion,
(2) partial ordering among method firing,
(3) firing prohibition of methods, and
(4) deadlock inevitability.
Refer to our previous paper [22] for the syntax and semantics of LPTL and the detail theory of verification and synthesis using Petri nets and LPTL.

3.4 Design Methodology

The proposed design methodology consists of five phases:
Phase 1: Design of elementary processes
Phase 2: Process interconnection and coordinator creation
Phase 3: Design of coordinators
Phase 4: Verification and adjustment
Phase 5: Code generation, execution and confirmation

First, Phase 1 and Phase 2 are carried out using directly MENDEL net because elementary processes are not so obscure. Then, the design of the coordinator whose structure may be obscure is done by stepwise refinement using the wide-spectrum causality matrix. A procedure of the matrix refinement is summarized as (step 1) **method recognition**; (step 2) **port and slot recognition**; (step 3) **state recognition**. Finally, MENDEL nets are verified and adjusted using LPTL, and executed visually. **Figure 7** outlines the flow chart of the following design methodology.

[Phase 1]
(Step 1) Find elementary processes:
Find all elementary processes whose structure is well defined, and enumerate the methods, slots, and ports for each process. Most hardware-constrained processes are elementary. For example, "cage" is one of the elementary processes in a lift system, and its methods include "move_up", "move_down", "open_door", and "close_door".
(Step 2) Construct a MENDEL net for each elementary process:
Construct a MENDEL net for each elementary process by appending the state elements and arrows to the methods, slots, and ports listed in step 1. Then, classify each port and method as external one, which becomes a plug and is accessed from other processes, or internal one.

[Phase 2]
Interconnect elementary processes and create a coordinator:
Create a new process (parent process) that consists of the elementary processes (subprocesses). Interconnect the plugs of these subprocesses with asynchronous communication and synchronous communication. Some plugs which cannot be directly connected may remain; in this case some process is required to coordinate them. Create a new subprocess (the coordinator) and connect the remaining plugs to it. Note that an initial coordinator has only external ports and methods that are connected to the remaining plugs of the elementary processes.

[Phase 3]
(Step 1) Create an initial causality matrix and recognize methods:
Create an initial causality matrix of the coordinator, in which only external ports and methods, listed in phase 1, are filled. Then, recognize all functions (method candidates) of the coordinator and fill them in the matrix.
(Step 2) Recognize internal ports and slots:

Judge existence of the causality relation between external / internal methods and external ports, and fill abstract level judgment (+ or blank) in the matrix. Stepwise refine these causality relations into a less abstract level. This stepwise refinement helps a designer to recognize additional internal ports and slots which are necessary to refine the causality relations.
(Step 3) Recognize state elements:
Find logical state elements of the coordinator (e.g., active, sleep, waiting, busy, etc.), and add them and fill causality relations in the matrix. In the causality relation, each method should be decided whether it is enabled or not in each logical state. In addition, consider the partial ordering of method firing, and introduce dummy control state elements to put method firing in order.
(Step 4) Describe KL1 codes:
Describe the detailed condition and action for each method by KL1. At this point, the causality matrix reaches its most concrete level.
(Step 5) Generate MENDEL net processes:
Finally, generate MENDEL net processes of coordinators straightforward from the most concrete-level matrices.

In addition, the following design rules are applicable during Phase 3.
(Rule 1) Cause design backtracking
When any design failures or unexpected functions that require structural rearrangements are detected, go back to any previous steps in Phase 3.
(Rule 2) Refine methods, slots, and ports:
Decompose and modify methods, ports, and slots if they have compound functions or meanings.
(Rule 3) Decompose coordinators:
Decompose coordinators if the coordinator becomes too large or too complex.

[Phase 4]
(Step 1) Verification:
Verify and analyze whether the constructed MENDEL net satisfies the given LPTL constraints (e.g., deadlock-free, interlock, etc.).
(Step 2) Adjustment:
If the MENDEL net does not satisfy all the specified LPTL constraints, the designer must adjust the MENDEL net to satisfy them, manually using a MENDEL net editor or automatically using the theorem proving method of LPTL.

[Phase 5]
Code generation, execution and confirmation:
Finally, compile the MENDEL net into a KL1 program, execute it, and confirm that it works well (i.e., it satisfies your requirements).

4. MENDELS ZONE

MENDELS ZONE is a CASE tool kit for concurrent programs. The target concurrent programming language is MENDEL [13], which is a textual form of a MENDEL net. Furthermore, MENDEL programs are compiled into the concurrent logic programming language KL1 [12] and executed on the parallel

computer Multi-PSI [16]. MENDEL is regarded as a user-friendly macro language of KL1, whose purpose is similar to A'UM [17] and AYA [18]. However, MENDEL (MENDEL net) is more convenient for programmers to use in designing cooperating discrete event systems. MENDEL programs can also be translated into the C language and be executed on a distributed personal computer system [19]. MENDELS ZONE provides the following CASE tools:

– an automatic generation tool of MENDEL processes from algebraic specification [20],

– a MENDEL-net-based programming environment, and

– a performance evaluation tool [21].

This paper focuses on the MENDEL-net-based programming environment (**Fig.8**), which supports the design methodology described in the previous section.

(1) Graphic Editor (**Fig.8** (a))
The designer constructs each process of the MENDEL net using a graphic editor which provides the creation, deletion, and placement functions for ports, state elements, methods, arrows, and tokens. This editor also supports the expansion and reduction of nets, and the transition over the process-oriented hierarchy (i.e., from process to subprocess, and vice versa).
(2) Method Editor
The method editor provides several editing functions specific to a high-level Petri net. Using the method editor, the designer describes methods (their conditions and actions) in detail using KL1.
(3) Process Library (**Fig.8** (b))
Reusable processes are stored in the process library. This library tool supports browsing and searching.
(4) Causality Matrix Editor
This spread-sheet editor supports stepwise refinement of the causality matrix. It provides the following functions:
- creation, deletion, and renaming of methods, ports, slots, and state elements,
- dividing methods, ports, slots, and state elements into detailed ones,
- checking whether refined relations are legal, and
- localizing and focusing the view of relations of designer's interest.

(5) Verification and Adjustment Tool (**Fig.8** (c))
First, only skeletons of MENDEL net structures are automatically retracted (detailed KL1 codes of methods are ignored) since our implemented verification and adjustment tools are only applicable to bounded nets. The verification tool checks whether a MENDEL net satisfies the given LPTL constraints entered by the designer using the LPTL editor [22]. If the net fails to satisfy the constraints, the adjustment tool can automatically adjust (tune up) the net to satisfy the LPTL constraints by adding an arbiter process [23]. The verification and adjustment are based on the theorem proving method of LPTL that is efficiently executed on Multi-PSI.
(6) Program Execution on Multi-PSI
The adjusted MENDEL net is translated into its textual form (MENDEL program). The MENDEL program is compiled into a KL1 program, which can be

executed on Multi-PSI. Each Process of the MENDEL program can run on the different CPU. Several compilation techniques (e.g., separate compilation) are introduced here to deal with large-scale programs. During execution, firing methods blink on the graphic editor, and the values (colors) of the tokens are displayed on the message window (**Fig.8** (d)). The designer can visually check that the program behaves satisfactorily.

5. EXAMPLE: LIFT SYSTEM

5.1 Problem

This problem is a revised version of the popular problem presented for the 4th International Workshop on Software Specification and Design [24].

Lift System:
One lift is to be installed in a building with "M" floors. The problem concerns the logic to move cages between floors according to the following constraints:
(1) The cage has a set of buttons, one for each floor. These illuminate when pressed and cause the cage to visit the corresponding floor. The illumination is cancelled when the corresponding floor is visited by the cage, or when the button is pulled out (cancelling the request).
(2) Each floor has two buttons, one to request an up-lift and another to request a down-lift. These illuminate when pressed. The illumination is cancelled when a cage visits the floor and is either moving in the desired direction, or has no outstanding requests. The illumination may also be cancelled by pulling the button out (cancelling the request).

5.2 Observation of actual design process

The actual design process of the lift system will be traced using MENDELS ZONE. Here, "Pi-Sj:" means "Step j of Phase i"; it is an index for the design methodology.

(1) **P1-S1:** This lift system has four elementary processes: "cage", "button", "floor_button_panel", and "cage_button_panel".
(2) **P1-S2:** The constructed MENDEL nets of these elementary processes are shown in **Fig. 9**. "floor_button_panel", and "cage_button_panel" are constructed as parent processes of "buttons". However, since they are very physical and require no coordinators, they ware regarded as elementary processes. Remark, c = cage, f = floor, req = request, can = cancel, vis = visit.
(3) **P2**: The top level "lift_system" is constructed by interconnecting 3 subprocesses: "cage", "floor_button_panel" and "cage_button_panel". Here are all plugs (open, up, down, c_req, c_can, c_vis, f_req, f_can, f_vis) remain unconnected. Therefore we create a coordinator process, and connect remaining plugs to it (**Fig. 10**).
(4) **P3-S1:** The initial causality matrix is created. It has 6 external ports and 3 external methods corresponding to the plugs initially connected in (3). We find

the following functions (method candidates) of the coordinator and enter them in the matrix:
- open: open the door of the cage when the cage visits the requested floor.
- up: move up to the target floor.
- down: move down to the target floor.
- c_req / f_req: accept requests from the cage_button_panel / floor_button_panel.
- c_can / f_can : delete the cancelled floor from the request queue.
- c_vis / f_vis : acknowledge the cage_button_panel / floor_button_panel when the cage arrives at the requested floor.

(5) **P3-S2:** Causality relations are entered at the abstract level. We recognize 3 internal slot candidates (req_que, current_f, target_f) and one additional method candidate (start), and add them to the matrix **(Fig. 11)**.
- req_que: a request queue in which all requests are stored.
- current_f: the number of the current floor that the cage is currently staying.
- target_f: the number of the target floor to which the cage will go.
- start: select a target floor from the request queue.

(6) **P3-R2&S3:** While refining the causality relations stepwise, we refine the operators and operands, and recognize new state elements. The following case shows a fragment of this refinement and state recognition process. When refining the operator c_can (**Fig. 12** (a), that is a local view of the matrix), we notice that there are two cases.
Case 1: The cancelled request remains in the req_que.
Case 2: The cancelled request has already been selected as the target_f.
Here, we divide c_can into c_can_1 (case 1) and c_can_2 (case 2), and try to refine each operator. Then, we recognize that the states (active and sleep) are necessary to refine c_can_2, and introduce them. Finally, the matrix shown in **Fig. 12** (b) is derived.

(7) **P3-R1:** We notice that we have missed the fact that the cage should stop and open the door at the floor that is stored in the req_que even if it is not a current target. Consequently, we must backtrack and modify the design; we must modify the method "open".

(8) **P3-R3:** The coordinator is divided into 2 coordinators ("cage_ controller" and "request_controller"), because the matrix becomes somewhat complex; so it is natural to divide its functions into the cage controller and the request controller. During this dividing, the following interface plugs (external ports) are introduced.
- start: acknowledge that the cage has started.
- end: acknowledge that the cage has visited the target floor.
- exit: acknowledge that the request has been cancelled.
- current: inform the current floor that the cage is staying.
- command: command that the cage opens the door or not.

Divide the matrix into new two matrices which corresponds to generated coordinator processes (cage_controller and request_controller).

(9) **P3-S3** (for cage_controller): After checking for any method conflicts, we introduce 9 dummy control state elements (active, sleep, etc.) to serialize the methods (current, command, open, pass, up, and down) to avoid the conflicts.

(10) **P3-S5:** We implement KL1 predicates used in the methods (e.g., cancel_rfl of **Fig. 12**) which completes the stepwise refinement of the matrix.

The final causality matrix of the cage controller is shown in **Fig.13**. From this matrix, MENDEL nets of the lift system can be generated automatically. **Fig. 14** shows the top level process ((a) lift_system) and one of the subprocesses ((b) cage_controller). In (b), slots are hided in the display for simplification, and a triangle represents OR-connection of synchronous communication.
(11) We can verify at its skeleton level whether or not this system satisfies the following constraints.
- Deadlock freedom of a cage:

LPTL formula = $GF(open \vee up \vee down)$
- Once a k-th floor is requested, the cage eventually visits the floor and opens the door unless the request is not cancelled:

LPTL formula = $G(((c\_req(k) \vee f\_req(k)) \wedge$

$G(\neg c\_can(k) \wedge \neg f\_can(k))) \supset F\ open(k))$

In this case MENDELS ZONE fortunately assures that this system satisfies these constraints in the skeleton level, so no adjustments are necessary.
(12) Finally, the KL1 program is generated from the MENDEL net and executed on Multi-PSI. We can confirm that the lift system works well.

5.3 Evaluation

We briefly evaluate our computer-supported methodology from our design experiences of the lift system. When we designed it using only a Petri net editor and with no methodology in the past days, we abandoned the use of the editor and did paper works in the earlier design phase because backtracking caused tedious editing and rearrangement. We used a Petri net editor only for a fair copy of the paper works. Experimental comparisons with other methodologies in case of using supporting environments will be future work, while paper-level comparisons will be done in the next section.

6. CONCLUSION AND RELATED WORKS

It has been shown that MENDELS ZONE is a Petri-net-based concurrent programming environment. MENDELS ZONE has several unique features: (1) a colored Petri net with a process-oriented hierarchy (MENDEL net), (2) compilation of the MENDEL net to a concurrent programming language and execution on a parallel computer, and (3) a design methodology for MENDEL nets and methodology support tools.

Using MENDELS ZONE, we have already constructed several small-scale discrete event systems, such as a machine control system for processing (i.e., etching) printed circuit boards [25]. In addition, we are just constructing and evaluating a large-scale control system for a power plant (about 4300 steps) [26]. For future work, we will extend the verification and adjustment abilities to manipulate detailed KL1 codes that are ignored in the current skeleton-level verification and adjustment.

We believe that existing competitors to MENDEL net and its design methodology are Real-Time Structured Analysis [27], PAISLey [28], Statemate [29], and G-LOTOS [14]. We have also proposed another design methodology using structured analysis [2]. All support data flow, state machine, and hierarchy. The most obvious difference from our approach is that these are not directly based on Petri nets. Each approach has similar abilities in general and distinct merits and demerits in detail. Nevertheless, we favor the Petri-net-based approach, because this approach can take advantage of graphical representations and a variety of analysis methods which have been and will be provided by many Petri net researchers.

Recently, other Petri-net-based design methodologies have been reported. Pinci and Shapiro proposed a methodology in which CPN are integrated with SADT (Structured Analysis and Design Technique) [30]. SADT is a sophisticated and well-used methodology for requirement analysis. However, it is based on data flow diagrams and lacks the state transition feature, while our methodology supports both data flow and state transition features. Etessami's rule-based design methodology [31] is another Petri-net-based approach which uses Abstract Petri Net (APN). A unique feature of APN is the combination of timed and colored Petri net. According to the rule-based design methodology, a designer first formalizes the specification by means of a set of rules and lists attributes representing the status of a target system. Then, he retracts places from the attributes and transitions from the rules, and finally describes APN. Etessami's approach seems to be in the same research direction as our approach. However, Etessami's methodology is less systematic (therefore, has no computer support) and is weaker in design backtracking. In addition, APN is not hierarchical.

Acknowledgment:
This research has been supported by ICOT. The authors would like to thank Ryuzou Hasegawa of ICOT for his encouragement and support. They are also grateful to Seiichi Nishijima and Yutaka Ofude of the Systems & Software Engineering Laboratory, TOSHIBA Corporation, for providing continuous support. They are also indebted to Fujio Umibe and Janet Gaboury for reviewing the original manuscript and suggesting revisions in its English.

REFERENCES

(1) N. Uchihira, et al. : "Concurrent Program Synthesis with Reusable Components Using Temporal Logic", Proc. COMPSAC'87 (1987).
(2) S. Honiden, et al. : "An Application of Structural Modeling and Automated Reasoning to Real-Time Systems Design", J. Real-Time Systems **Vol.1** (1990).
(3) N. Uchihira, et al. : "Synthesis of Concurrent Programs: Automated Reasoning Complements Software Reuse", Proc. 23rd HICSS (1990).
(4) "Special Issue: Programmable Controller" (in Japanese), KEISO **Vol.34**, No.23 (1991).

(5) K.Jensen : "Coloured Petri Nets: A High Level Language for System Design and Analysis", Advances in Petri Nets 1990, LNCS **483**, Springer-Verlag (1990).
(6) H.J.Genrich, K.Lautenbach : "System Modelling with High-Level Petri Nets", Theoret. Comput. Sci. **Vol.13** (1981).
(7) W.Reisig : "Petri nets and algebraic specification", Theoret. Compt. Sci., **Vol.80** (1991).
(8) P.Huber, et al. : "Hierarchies in Coloured Petri Nets", Advances in Petri Nets 1990, LNCS **483**, Springer-Verlag (1990).
(9) R.Milner : "Communication and Concurrency", Prentice Hall (1989).
(10) F.Feldbrugge : "Petri Net Tool Overview 1989", Advances in Petri Nets 1989, LNCS **424**, Springer-Verlag (1990).
(11) S.Kumagai : "Petri Net Tools" (in Japanese), J.SICE, **Vol.28**, No.9 (1989).
(12) T.Chikayama, et al. : "Overview of the Parallel Inference Machine Operating System (PIMOS)", Proc. FGCS'88 (1988).
(13) N.Uchihira : "Petri-Net-Based Concurrent Programming Language MENDEL: User's Guide", ICOT-TM (to appear) (1992).
(14) E. S. Lee, et al. : "Construction and Implementation of a Specification Environment SEGL Based on G-LOTOS" (in Japanese), J. IPSJ **Vol.32**, No.3 (1991).
(15) IMMOS Limited: "Occam 2 Reference Manual", Prentice Hall (1988).
(16) K.Taki : "The FGCS Computing Architecture", ICOT TR-460 (1989).
(17) K. Yoshida, T. Chikayama : "A'UM - Stream-Based Concurrent Object-Oriented Language -", FGCS88 (1988).
(18) K.Suzaki and T.Chikayama : "AYA: Process-Oriented Concurrent Programming Language on KL1" (in Japanese), KL1 Programming Workshop'91 (1991).
(19) N.Uchihira, S.Honiden : "Petri-Net-Based Concurrent Programming Language on Distributed Processing Systems", IEICE Technical Report, **Vol.89**, No.167 (1989).
(20) S.Honiden, et al. : "An Integration Environment to Put Formal Specifications into Practical Use in Real-Time Systems", Proc. 6th IWSSD (1991).
(21) S.Honiden, et al. : "An Application of Artificial Intelligence to Prototyping Process in Performance Design for Real-Time Systems", Proc. ESEC'91, LNCS **550** (1991).
(22) N.Uchihira and S.Honiden : "Verification and synthesis of concurrent programs using Petri nets and temporal logic", Trans. IEICE, **Vol.E73**, No.12 (1990).
(23) N.Uchihira, et al. : "Compositional Adjustment of Concurrent Processes to Satisfy Temporal Logic Constraints in MENDELS ZONE", IJPS SIG Notes SE-83-20 (1992).
(24) "Problem set", Proc. 4th IWSSD (1987).
(25) MENDELS ZONE, Demonstration explanation book of the national fifth generation computer system symposium (1991).
(26) MENDELS ZONE, Demonstration explanation book of FGCS'92 (1991).
(27) P. Ward : "The Transformation Schema: An Extension of the Data Flow Diagram to Represent Control and Timing", IEEE Trans. SE, **Vol.12**, No.2 (1986).

(28) P.Zave : "An Insider's Evaluation of PAISLey", IEEE Trans. SE, **Vol.17**, No.3 (1991)
(29) D.Harel,et al. : "STATEMATE: A Working Environment for the Development of Complex Reactive Systems", IEEE Trans. SE, **Vol.16**, No.4 (1990)
(30) V.O.Pinci, R.M.Shapiro : "An Integrated Software Development Methodology Based on Hierarchical Colored Petri Nets", Advances in Petri Nets 1991, LNCS **524**, Springer-Verlag (1991).
(31) F.S.Etessami, G.S.Hura : "Rule-Based Design Methodology for Solving Control Problems", IEEE Trans. on SE, **Vol.17**, No.3 (1991).

LEVEL	RELATION	EXPLANATION
Abstract level	+	some relation
	blank	no relation/undefined
Intermediate level (1)	<	input/cause
	>	output/effect
Intermediate level (2)	sync	synchronization
	rw	read or write from/to slot
	pp	pop or push from/to port
	trans	state transition
Intermediate level (3)	insync	synchronization with input data
	outsync	synchronization with output data
	read	read from slot
	write	write to slot
	pop	pop from port
	push	push to port
	in	reference of state
	from	transition from state
	to	transition to state
Concrete level	KL1	KL1 code

Table 1 Wide-spectrum causality relations

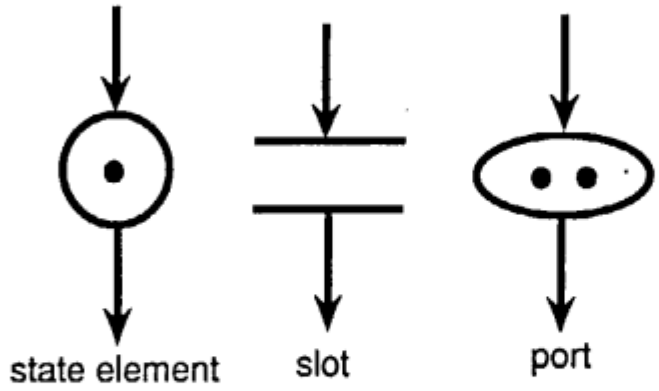


Fig. 1 Three types of places: state element, slot, and port

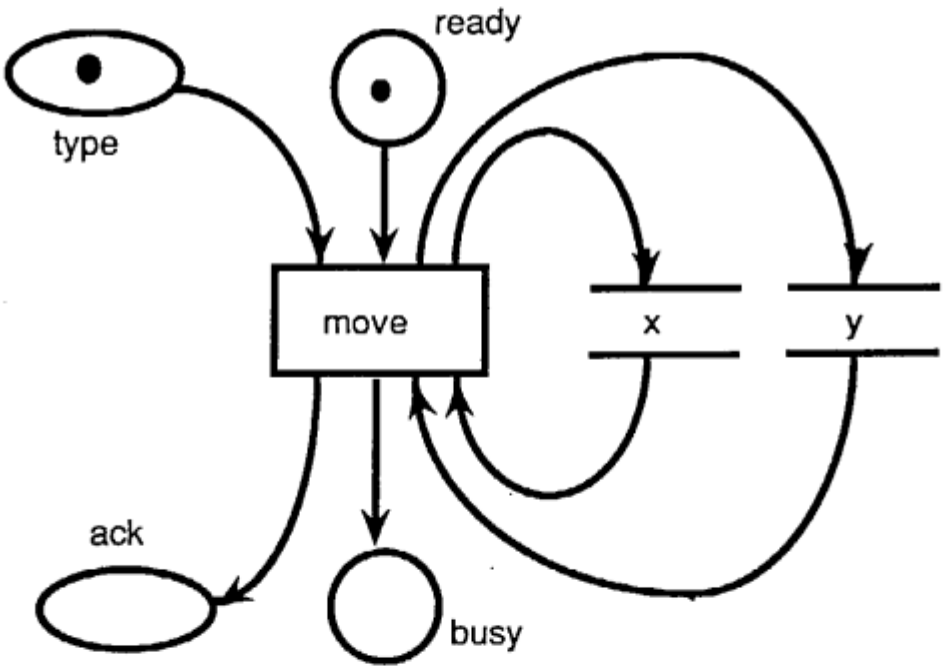
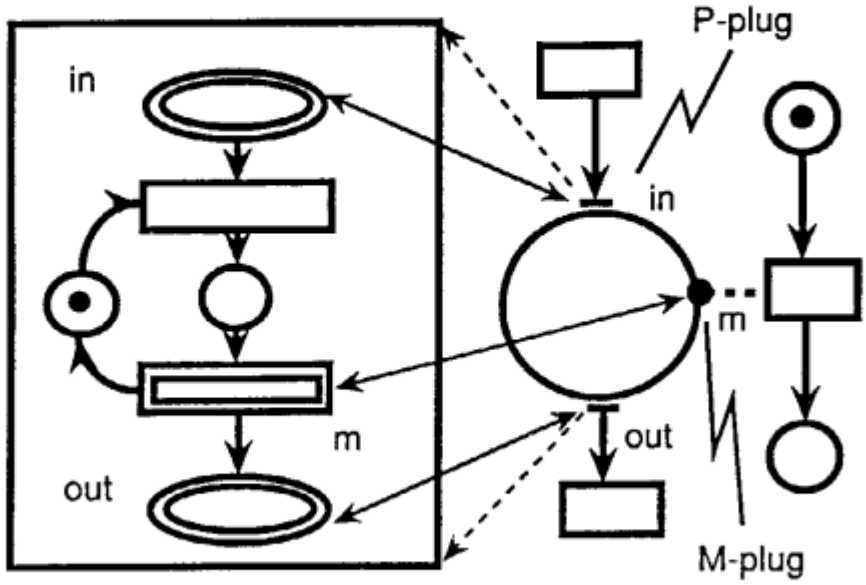
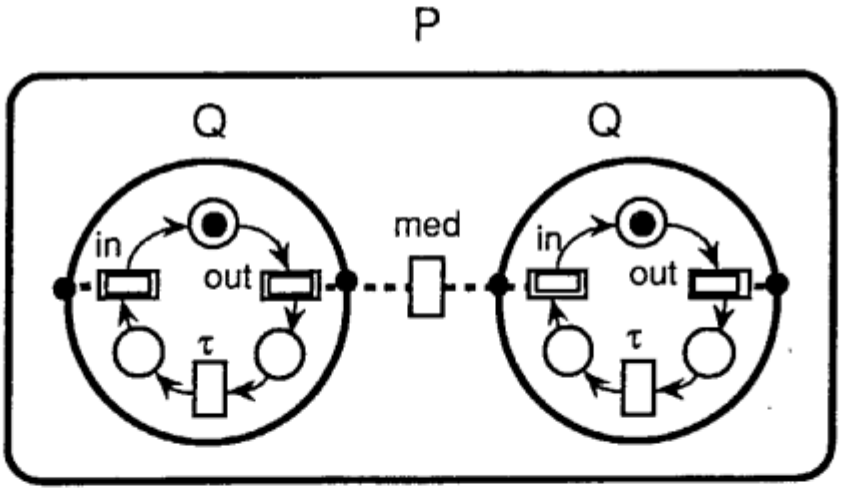


Fig. 2 Method example



(a)



(b)

Fig. 3 Process-oriented hierarchy

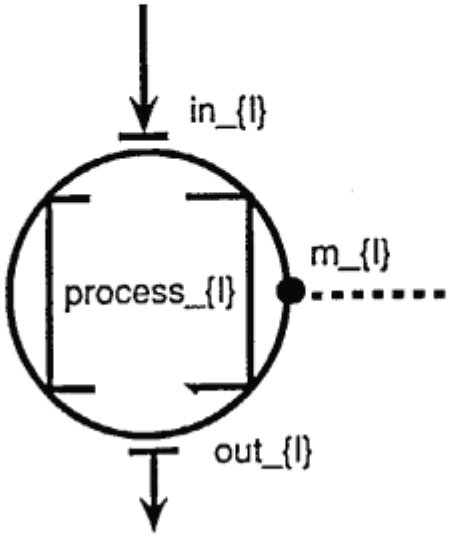


Fig. 4 Array representation

operand / operator	opd1	opd2	opd3	opd4		
	state		port		guard	body
		external				
opr1	+		push			
opr2	in		pop:X		X > 0	true
opr3		>	+	+		
opr4	+			+		
opr5	+	<	pp			

Fig. 5 Causality matrix

	free	busy	data	input	output		
	state	state	slot	port	port		
				external	external	guard	body
get	from	to	write:X	pop:X		X>0	true
put	to	from	read:X		push:Y	true	Y:=X+1

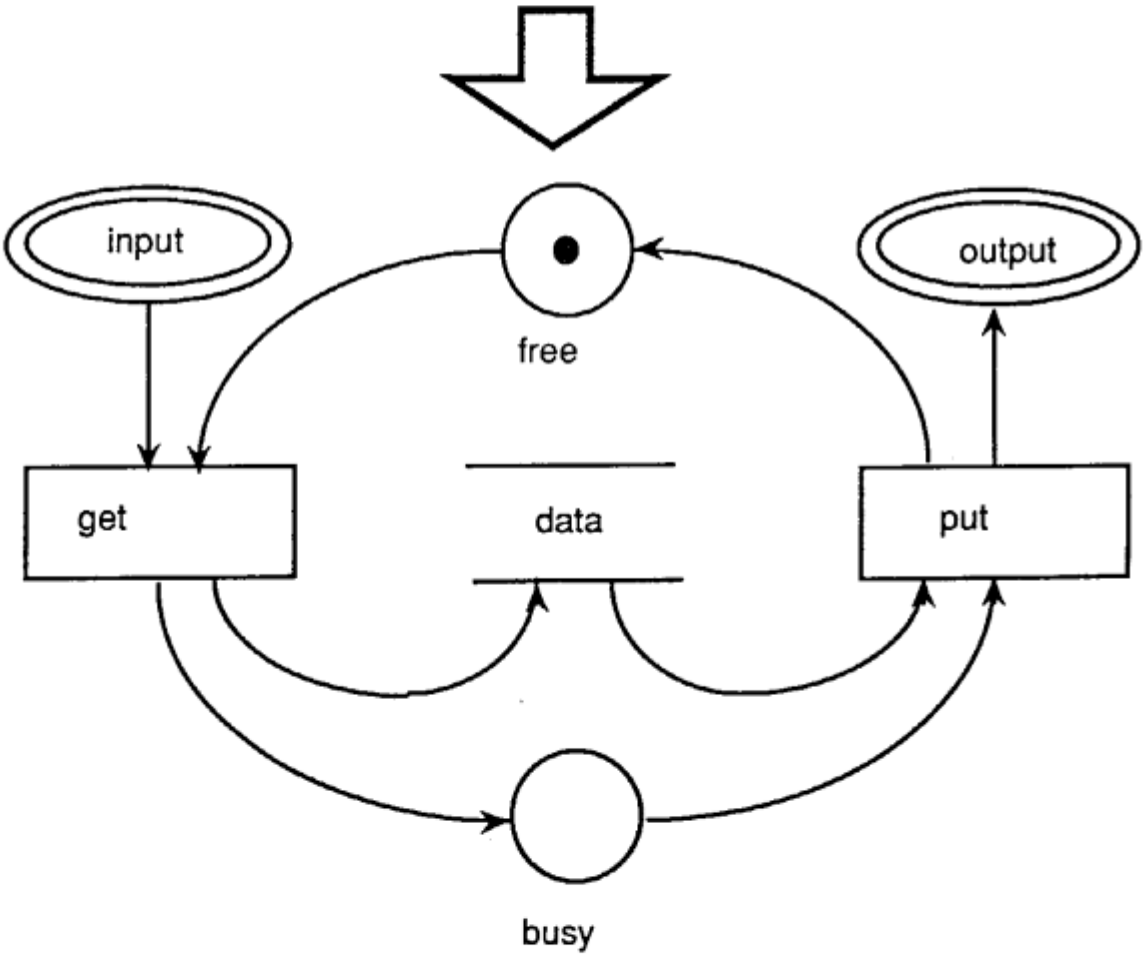


Fig. 6 Transformation from a matrix to a net

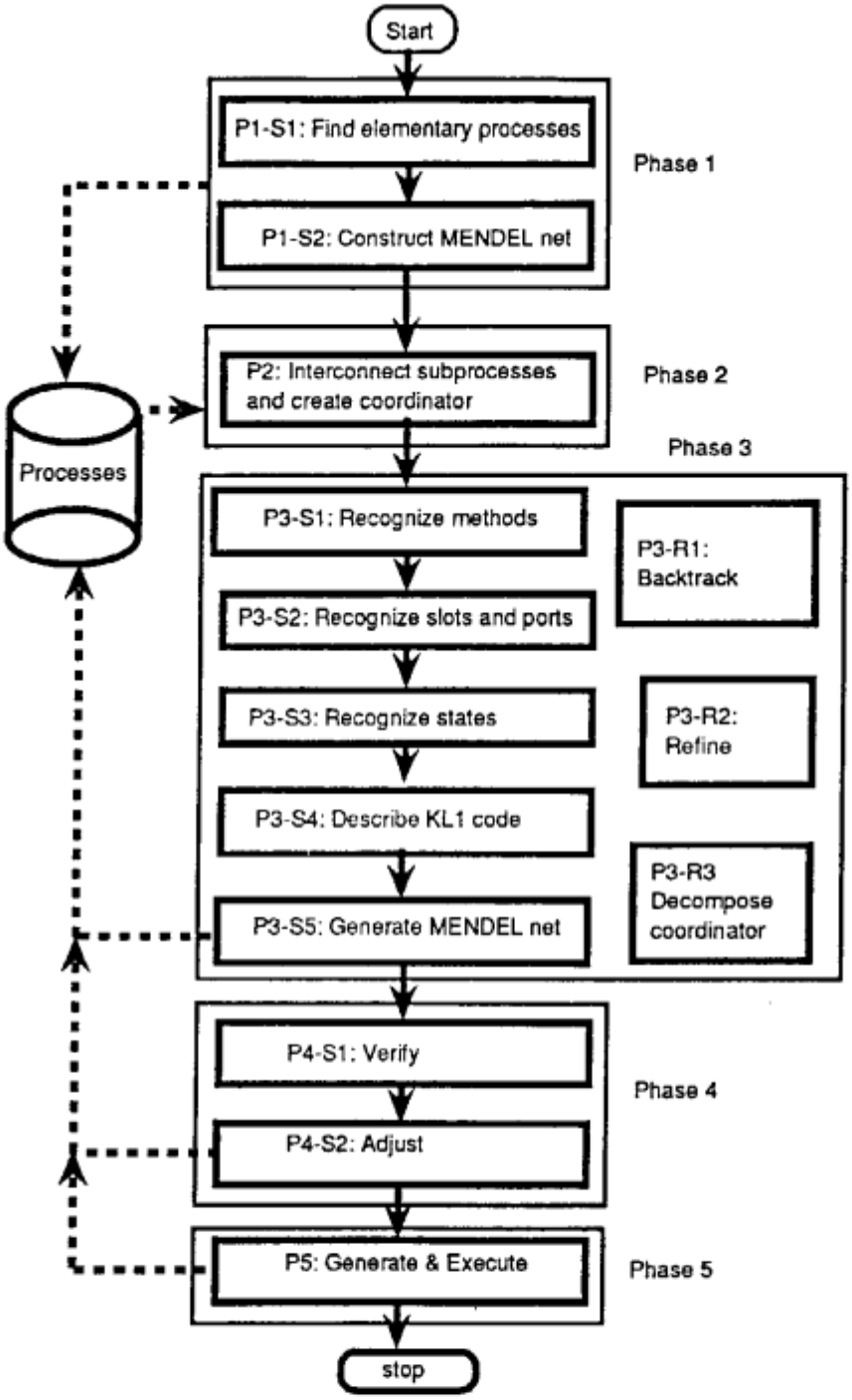


Fig. 7 Design methodology

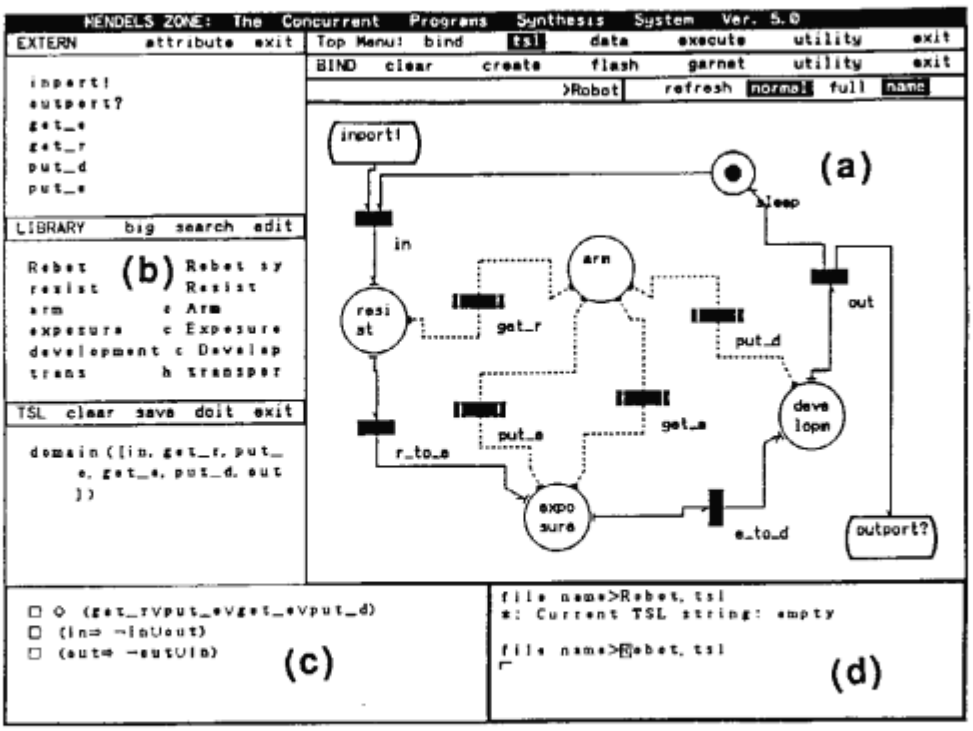


Fig. 8 MENDELS ZONE

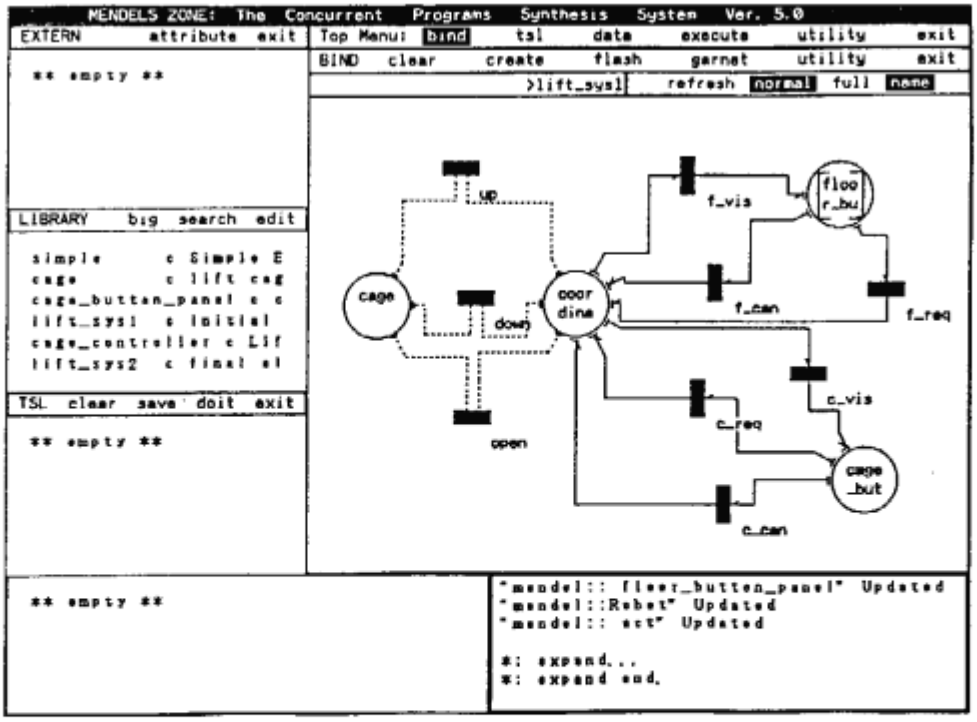
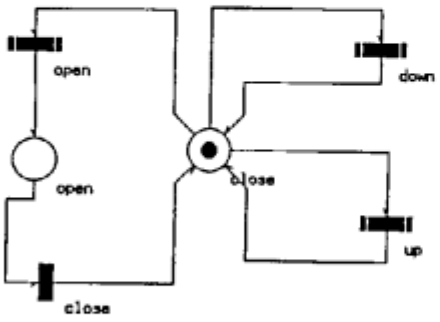
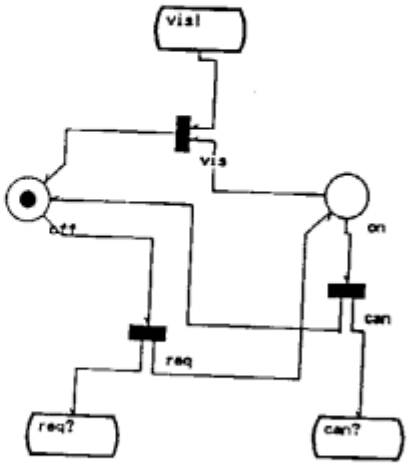


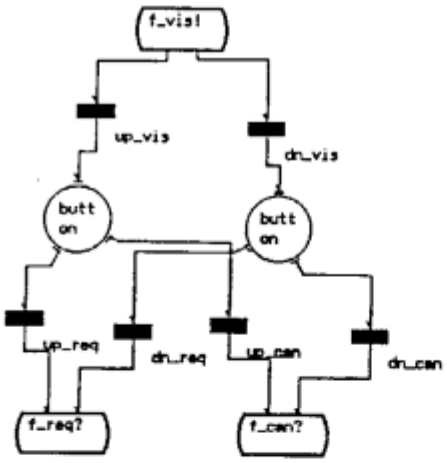
Fig.10 Process interconnection and coordinator creation



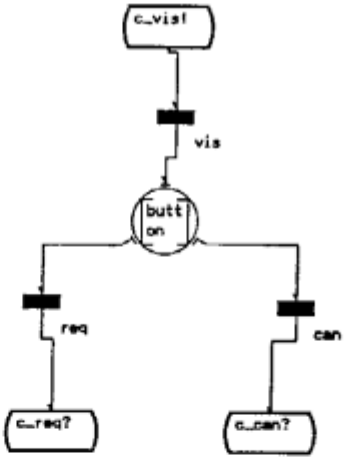
cage



button



floor_button_panel



cage_button_panel

Fig.9 Elementary processes of a lift system

	open	up	down	c_req	c_can	c_vis	f_req	f_can
	method	method	method	port	port	port	port	port
	external	external	external	external	external	external	external	external
open	+							
up		+						
down			+					
c_req				+				
c_can					+			
c_vis						+		
f_req							+	
f_can								+
f_vis								
start								

continue

f_vis	req_que	current_f	target_f		
port	slot	slot	slot	guard	body
external					
		+	+		
	+				
	+		+		
		+	+		
	+				
	+		+		
+		+	+		
	+	+			

Fig.11 Causality matrix (abstract level)

(a) Original matrix

	c_can	req_que	target_f		
	port	slot	slot	guard	body
c_can	pop	rw	read		

(b) Refined matrix

	c_can	req_que	target_f	active	sleep		
	port	slot	slot	state	state	guard	body
c_can_1	pop:X	read:Q1 write:Q2	read:Y	in		X¥=Y	cancel_rfl(X,Q1,Q2)
c_can_2	pop:X		read:Y	from	to	X=Y	true

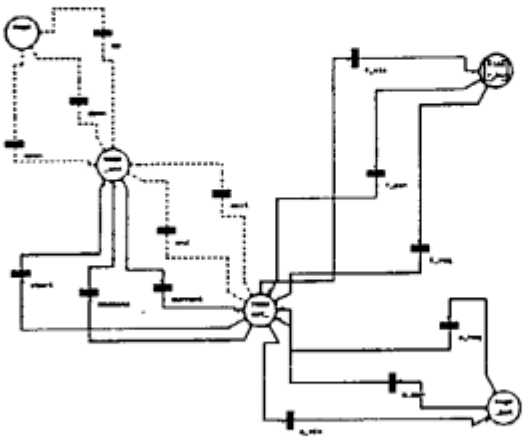
Fig. 12 Causality matrix
(refinement example)

	open	up	down	start	exit	end	command	current	sleep	active	c_floor
	method	method	method	port	method	method	port	port	state	state	slot
	external	external	external	external	external	external	external	external			
open1	outsync										
end						outsync			to		
exit					insync				to	from	
start				pop:X					from	to	
check1										from	read:(X,_)
check2										from	read:(X,_)
current								push:CF			read:CF
pass							pop:COM				
open2	outsync						pop:COM				
check3											read:(X,_)
check4											read:(X,_)
up		outsync								to	read:(X,_)
											write:(Y,up)
down			outsync							to	read:(X,_)
											write:(Y,down)

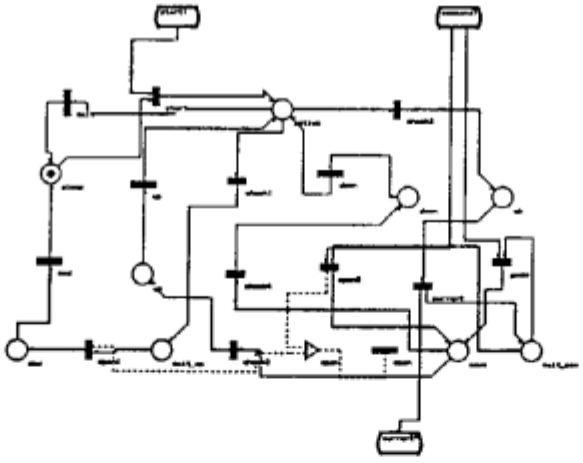
continue

target_f	wait_op	end	ch	wait_com	move	up	down		
slot	state	state	state	state	state	state	state	guard	body
	from	to						TRUE	TRUE
		from						TRUE	TRUE
								TRUE	TRUE
write:X								TRUE	TRUE
read:Y	to							X=Y	TRUE
read:Y			to					X\=Y	TRUE
			from	to				TRUE	TRUE
				from	to			COM=pass	TRUE
				from	to			COM=open	TRUE
read:Y					from	to		X<Y	TRUE
read:Y					from		to	X>Y	TRUE
						from		TRUE	Y:=X+1
							from	TRUE	Y:=X-1

Fig.13 Causality matrix (concrete level)



(a) lift_system



(b) cage_controller

Fig.14 Constructed MENDEL net