ICOT Technical Report: TR-449

TR-449

A Concurrent Program Synthesis Using Petri Net and Temporal Logic in MENDELS ZONE

by
N. Uchihira, H. Kawata & S. Honiden

January, 1989

ICOT
Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan
(03) 456-3191－5
Telex ICOT J32964

Institute for New Generation Computer Technology

A CONCURRENT PROGRAM SYNTHESIS USING PETRI NET AND TEMPORAL LOGIC IN MENDELS ZONE

Naoshi Uchihira
Hideji Kawata*
Shinichi Honiden

Systems & Software Engineering Lab.
TOSHIBA Corporation
Yanagicho 70, Saiwai-ku, Kawasaki 210, JAPAN
PHONE: (044) 548 - 5465

*His current address: ICOT reserch lab.

Keyword: Concurrent Program Synthesis, Automatic Programming, Temporal Logic, Petri Net, Prolog, Concurrent Program Language, Software Reuse, Environment

ABSTRACT

MENDELS ZONE is a concurrent programming environment, which supports program synthesis with reusable components, using Petri net and temporal logic. The target concurrent programming language is MENDEL/88, in which objects run concurrently and communicate with each other through streams. A MENDEL program has body parts and synchronization parts. In MENDELS ZONE, the program synthesis consists of two major steps: (1) the body part construction from reusable software and (2) the synchronization part synthesis from temporal logic specification. MENDEL net, which is a restricted Petri net automatically generated from MENDEL body parts, is also used in step (2) of the above, in order to synthesize synchronization parts consistent with body parts. In this paper, we will describe mainly the synthesis of the synchronization part .

1. INTRODUCTION

The two major purposes of program synthesis and automatic programming are to increase software productivity and to generate a program that is assured of being correct. Recently, software reuse is expected to greatly increase software productivity, and many program synthesis method based on software reuse have been presented. However, most of them are only for sequential programs and not for concurrent programs. In concurrent programs, correctness is very important, as it is not easy for a human being to make a correct synchronization part of the concurrent program. Therefore, verification and synthesis on concurrent programs using logic, especially temporal logic, have been studied for a long time to assure correctness.

In Manna and Wolper [1,2], Propositional Temporal Logic (PTL) is used for concurrent program synthesis. They show a theorem proving method which can synthesize synchronization parts of a concurrent program using PTL or extended temporal logic (ETL). In this method, a model graph, generated in the PTL decision procedure, is considered as a state transition diagram for processes. CSP program codes which execute synchronization are generated from this state transition diagram. Also, some other works [3,4] are done on synthesis using temporal logic.

One of the practical approaches to automatic programming in large scale applications is a program synthesis which utilizes an automated reasoning mechanism with software reuse. Automated reasoning approach can only synthesize small scale programs and cannot support large scale applications. In Manna and Wolper's synthesis method, a synchronization part is generated automatically. However, the other part, namely, a body part in our method, must be created by the programmer. We propose a new synthesis method, which is a combination of software reuse and an extension of Manna and Wolper's method. Our method consists of two major steps: (1) body part construction by interconnecting reusable components and (2) synchronization part synthesis from temporal logic specification. Since the synchronization part has a great deal to do with the body part, the synchronization part should be synthesized under constraints of the body part structure. In our method, the body part is represented by a restricted Petri net. Therefore, the synchronization part is synthesized from the temporal logic specification and Petri net. The target language is MENDEL/88. MENDEL/88 is a Prolog-based hierarchical concurrent programming language, which is an extended language of MENDEL [5].

MENDELS ZONE is a concurrent program synthesis environment which supports our method. MENDELS ZONE has been implemented on the Prolog Machine, which is visually similar to the CASE tool based on Real Time Structured Analysis by Ward[6], because MENDEL is a hierarchical stream-based language.

In this paper, we will first provide our model of a concurrent program, then proceed to explain MENDEL/88, the language which is based on this model. After mentioning briefly MENDELS ZONE, the synchronization part synthesis will be mainly described. Please note that the body part construction has already been described in another paper[7]. We will show a synthesis example, called "Hierarchical Dining Philosophers", in the last part of our paper.

2. CONCURRENT PROGRAM MODEL

This section provides a concurrent program model, using Petri net and temporal logic. This model is the semantical base of MENDEL/88.

Petri Net

Petri net has been extensively used to model the concurrent systems. As Petri net is a general and essential schema, it is possible to represent the dynamic behavior of most concurrent systems. We base our concurrent program model on Petri net.

Transition Control

The dynamic behavior of the concurrent systems has two aspects, one is a data-driven aspect (represented by data flow) and the other is an event-driven aspect (represented by control flow). In Petri net, all behaviors are represented by only data-driven transitions. In Petri net, Data flows and control flows are treated in the same manner. However, it is not suitable for general cases because a net may be too complicated with many arcs that act as control flows.

Some of the control flows can be substituted by controlling transition firing (i.e. determination of which transition is fired) in meta-level. A "transition control program" is used to manage this meta-level control. According to this program, only one enabled transition is selected out of all enabled transitions, and then fired. This meta-level control makes a net simpler and focuses the net on data flows. Here, we can regard the transition control as the synchronization supervisor, the transition

control program as the **synchronization part** of a concurrent program, and the net focused on data flows as the **body part**.

Temporal Logic

Most of transition control are local and prohibitive. As a result, it is easier to give a transition control program by declarative description, rather than by procedural description. In our model, a transition control program is specified by the temporal logic. A temporal logic specification is translated into a procedural program by automated reasoning mechanism. A relation of Petri net and temporal logic on our model is illustrated in Fig.1.

Hierarchical Structure

A pure Petri net is flat, and has no hierarchical structure. Our model should have the hierarchical structure as the actual concurrent system has. Therefore, the hierarchical structure is introduced in our model, where transition control is executed at each hierarchical layer. This means each layer has one transition control program (Fig.2). Each hierarchical component is called an "object", by which we mean an autonomous agent.

Software Reuse

Each object, which has a local Petri net and a transition control program, become a reusable component. A new object can be synthesized with reusable objects. To be more precise, a body part is first composed by interconnecting reusable objects with transitions and arcs, and then a transition control program, that is a synchronization part, is provided on the body part as illustrated in Fig.3. In the software reuse environment, the synchronization mechanism should be separated from reusable objects, such as in a path expression [8]. The transition control program of our model satisfies this requirement.

3. MENDEL/88

A MENDEL program is identical to a MENDEL object, which is either an atomic object or a compound object.

a. Atomic Object

(Syntax)

An atomic object consists of declaration, method, and junk parts.

```
<atomic object> ::=
        atomic object <object name> :{
                dec : {                 <declaration part>      } ;
                meth : {        <method part>           } ;
                junk : {        <junk part>             } ;
        }.
```

Declaration part

The declaration part includes four items :

1) External input ports

```
        inport( <port name>, ... ) ;
```

2) External output ports

```
        outport( <port name>, ... ) ;
```

3) Internal state variables

```
        state( <port name>:[<state>,...] ! <initial value>,...) ;
        openstate( <port name>:[<state>,...] ! <initial value>,...) ;
```

4) Internal data variables
data(<port name>!<initial value>,...) ;

Generically we call them "ports". Specifically, we call 1) and 2) external ports, and 3) and 4) internal variables.

Method part
The method part includes several methods. Each method is generally presented as follows:

method (<port name> ? <term>, ... ,<port name> ! <term>, ...)
← <guard> | <Prolog goals> ;

Junk part
<Prolog clauses>

(Semantics)
An atomic object can be regarded as a process. An atomic object has several external I/O ports and internal variables, which is handled by methods.

External I/O port:
Through this ports, objects can transmit messages from/to the outside.

Internal variable:
Internal state variables characterize a state of the object. For each internal state variable, a domain should be previously determined by **[<state>,...]**. The "openstate" declares internal state variables which can be referred from the outside. Variables other than state variables are internal data variables, that are accessible only from inside of the object itself.

An atomic object has several methods. In this method description, **"?"** means "input from a port" and **"!"** means "output to a port", such as CSP [9]; "|" means commitment operator, such as GHC [10].
For example: method(age?N, place?P, alcohol!T) ← N > 19, P= japan | T = ok ;

One of methods is selected, which satisfy the following conditions:
(1) Each of the terms after **<port name>?** can be unified with a received message from the external port or a value in the variable.
(2) All Prolog predicates in <guard> succeed.
When the method is selected, all Prolog predicates in **<Prolog goals>** are evaluated. Each term after **<port name>!**, which has been unified in **<Prolog goals>**, is sent as a message to the external port, or assigned to the variable. Prolog predicates written in **<guard>** have no side effect. If no method satisfies above conditions, the object is suspended. This method selection mechanism is similar to Dijkstra's guarded command.

The junk part includes some Prolog clauses, which may be called by methods or other Prolog clauses.

b. Compound Object

(Syntax)
<compound object> ::=
object <object name> :{
dec:{ <declaration part> } ;
body:{ <body part> } ;
sync:{ <synchronization part> } ;

}.

Declaration part

The declaration part includes 5 items :

1) External input ports
```
inport( <port name>, ... ) ;
```
2) External output ports
```
outport( <port name>, ... ) ;
```
3) Message gates
```
mgate( <gate name>,...) ;
```
4) Signal gates
```
sgate( <gate name>,...) ;
```
5) Free gates
```
fgate( <gate name>,...) ;
```

Body part

```
body :{
        innode(<port name>?[<gate name>,...],...) ;
        outnode(<port name>! [<gate name>,...],...) ;
        <object name>(
                <open state variable name>:[<state>, ... ],...
                        <port name>! [<gate name>, ... ],...
                        <port name>?[<gate name>, ... ],...) ;
                        ......  ......  ......  .......  ......  .....
} ;
```

Synchronization part

The synchronization part is represented by a finite automaton, as follows:

```
sync:{
                (       [<state transition rule>,...], <initial state>          )
} ;
<state transition rule> ::=
        trans(<current state>,<next state>,<guard condition>,<gate name>)
```

(Semantics)

A compound object has several external ports, similar to an atomic object. A body part of the compound object consists of several objects. These objects are either atomic or compound. This forms a hierarchical structure of objects. External ports of component objects are interconnected with streams in the body part. Messages between objects are transmitted through these streams. The stream is a one-to-one asynchronous one-way path. Each stream has one unique gate, therefore, streams are identified by gate names. For example, if outport A has the same gate name as inport B in the body part (i.e. obj1(A?[g1]), ... obj2(B![g1]), ...), this means that A and B are connected with a stream, through which messages are transmitted from outport A to inport B. Both the **innode** and **outnode** are interfaced to the outside of the compound object. Gates are used for control of messages in streams.

In MENDEL/88, a simple synchronization mechanism is provided by a method selection mechanism. The object is suspended until it receives all required messages. However, the above mechanism is so simple that for a complicated synchronization, it requires complicated stream interconnecting. (Many of these stream are only for control, not for data.) Therefore, an additional synchronization mechanism using gate is

introduced. Each stream has one gate, which controls message streams. The gate opens to let only one message pass through. When there is no message in the stream, the gate cannot be opened. In the following section, the gate will be formalized as a transition of Petri Net. A sequence of opened (fired in Petri net) gates can be regarded as a formal language, therefore, the gate control rules are represented by a finite automata with guard conditions in the synchronization part. For example, trans(s1,s2,[g1=full, g2=empty], g1) means if the current state is s1 and "g1=full & g2=empty" is true, then it is possible to fire g1 and move to the new state, s2.

The signal gate is a gate that is connected to either inport (called input signal gate) or outport (called output signal gate). This gate is mainly used as a signal generator to activate objects. The free gate is a gate which is out of the transition control. The free gate can be opened at any time. Therefore no free gate appears in the synchronization part. A simple example of the MENDEL/88 program is shown in Fig.4, which computes factorial numbers.

4. MENDELS ZONE: MENDEL PROGRAM SYNTHESIS ENVIRONMENT

MENDELS ZONE is a visual programming environment for MENDEL. MENDELS ZONE supports rapid software prototyping, which includes program synthesis method, illustrated in Fig.5 The program synthesis method of MENDELS ZONE consists of five steps:

(Step1) Create MENDEL atomic objects and register them on a object library.
(Step2) Construct the body part of compound objects by software reuse.
(Step3) Synthesize the synchronization part of compound objects from temporal logic specification.
(Step4) Execute the synthesized program and test the program visually.
(Step5) Register the synthesized compound object on a library.

The step on the body part construction has already been proposed in our former works [7]. In this step, a body part is automatically constructed from reusable objects by the object interconnection mechanism using semantic network. In this paper, we will describe in detail another important step -- the synchronization part synthesis. A specification of the synchronization part is written by TSL (Temporal Specification Language), which is a specification language based on propositional temporal logic. The MENDEL net generated from MENDEL body parts, is also used to synthesize synchronization parts which are consistent with body parts. In the following section, we will begin with MENDEL net and TSL, and then proceed to describe how a synchronization part can be synthesized using TSL and MENDEL net.

We have already implemented MENDELS ZONE on a Prolog machine, which includes the MENDEL interpreter. MENDELS ZONE provides five windows for user interface (Fig.6): (1) System window, (2) Object library window which displays MENDEL reusable objects, (3) I/O port window which shows external I/O ports, (4) TSL window which shows a temporal logic specification for a MENDEL synchronization part, and (5) Graphic programming editor window in which we can edit and construct a MENDEL body part.

5. MENDEL NET

A schema of MENDEL program can be viewed as a restricted Petri net, which is called MENDEL net.

Definition

A MENDEL net is specified by the following elements:

OBJ: A finite set of *objects*

IP_i ($i \in OBJ$): A set of *inner places* in object i.

PP_i ($i \in OBJ$): A set of *port places* in object i.

We use P to denote $\bigcup_{i \in OBJ}(PP_i \cup IP_i)$.

MT_i ($i \in OBJ$): A set of *method transitions* in object i.

GT : A set of *gate transitions* .

We use T to denote $\bigcup_{i \in OBJ} MT_i \cup GT$.

A : A set of *arrows*. Each arrow is a pair (p,t) or (t,p), such that $p \in P$, $t \in T$.

$P_{in}(t) \subset P$: A set of places, which have input arrows to transition t. For each place $p \in P_{in}(t)$, $a=(p,t) \in A$.

$P_{out}(t) \subset P$: A set of places, which have output arrows to transition t. For each place $p \in P_{out}(t)$, $a=(t,p) \in A$.

Here, if $t \in GT$, then $P_{in}(t)$ and $P_{out}(t)$ have maximum one element respectively, and $P_{in}(t) \cup P_{out}(t)$ is not empty. For each $t \in MT_i$, if all $p \in P_{in}(t)$ have a *token*, the method transition t is enabled. For each $t \in GT$, if all $p_{in} \in P_{in}(t)$ have a token and all $p_{out} \in P_{out}(t)$ have no token, the gate transition t is enabled. This is a special rule of MENDEL net. When transition t fires, one token is subtracted in each $p \in P_{in}(t)$ and one token is added in each $p \in P_{out}(t)$, in the same way as in the Petri net.

Representation by Diagrams

MENDEL net is represented by diagrams almost similar to Petri net. The basic conventions are as follows:

- Each inner place is represented by a circle  .
- Each port place is represented by a rectangle  .
- Each gate transition is represented by a bold bar  .
- Each method transition is represented by a fine bar .
- Each object is represented by enclosing places and transitions belonging to the object with a line "_".

Transformation of MENDEL Program to MENDEL Net

A MENDEL program can be transformed to MENDEL net automatically, in the following manner:

- Individual inner variables and external I/O ports are transformed to inner places and port places respectively. Particularly for state variables, which is a kind of inner variables, they have different places for different states. An example is a state variable **flag** which takes two states: **on** and **off**. They are transformed to two inner places, **flag=on** and **flag=off**.
- Individual gates and methods are transformed to gate transitions and method transitions respectively.
- Arrows in MENDEL net are generated according to the following rules:
  (We denote gate and method by t*, I/O external port and inner variable by p* in a MENDEL program, which are transformed to transition t and place p in MENDEL net respectively.)

(R1) If an output external port p_i^* of object i and an input external port p_j^* of object j are connected by gate g*, then $P_{in}(g)=\{p_i\}$ and $P_{out}(g)=\{p_j\}$.

(R2) If a method m* has input ports ($ip_1^*, ip_2^*, \ldots, ip_k^*$) and output ports ($op_1^*, op_2^*, \ldots, op_n^*$), then $P_{in}(m)=\{ip_1, ip_2, \ldots, ip_k\}$ and $P_{out}(m)=\{op_1, op_2, \ldots, op_n\}$ in principle. Additionally, there are some special modifications: In the case of a data variable p*, if $p \in IP_i$ and $p \in P_{in}(m)$, then add p to $P_{out}(m)$. In the case that a state variable s has domain D, if $s=x \in IP_i$, $s=x \in P_{out}(m)$, and $s=y \notin$

$P_{in}(m)$ for some state x in D and any state y in D, then add s=y to $P_{in}(m)$ for all states y in D. A MENDEL net of the program in Fig.4 is shown in Fig.7.

6. TEMPORAL SPECIFICATION LANGUAGE

This section introduces a specification language for a synchronization part of MENDEL program, called Temporal Specification Language (TSL). A synchronization part of a MENDEL compound object can be synthesized from a specification written by TSL.

(1) Definition
TSL is based on a linear time propositional temporal logic (PTL). It differs from PTL in having single-state conditions for each element. Also, TSL syntax is specialized as a specification language of the synchronization part. TSL is defined as follows:

Syntax
The set of formulas are defined from a finite set of elements E and a finite set of states S inductively as follows:
- If e E and $s \in D \subset S$, then e(s,D) is a formula. D is called the domain of element e.
- If f1 and f2 are formulas, then ~f1, f1 &f2, f1#f2, f1=>f2, [ ]f1, <>f1, @f1, and f1 $ f2 are formulas.

For example, if switch $\in$ E and D={on, off},
[](switch(on, {on,off}) => @switch(off, {on,off})) is a TSL formula.

Semantics
Intuitively, operators have the following meanings:
~ : NOT, & : AND, # : OR, => : IMPLY, <=>: EQUIVALENT
[ ]f (read always f) : f is true for all future states,
<>f (read eventually f) : f is true for some future state,
@f (read next f) : f is true for the next state,
f1 $ f2 (read f1 until f2) : f1 is true until f2 becomes true.
TSL has the same semantics of PTL with a single-state conditions. TSL formulas are transformed to PTL formulas with a single-state condition:

$$\bigwedge_{e \in E} ((\bigvee_{s \in D} e(s,D)) \wedge \bigwedge_{s_1,s_2 \in D, s_1 \neq s_2} \sim(e(s_1,D) \wedge e(s_2,D))),$$

and then interpreted in PTL semantics. A single-state condition means each element has only one state in its domain each time. It is an extension of a single event condition in [2].

(2) Model of TSL Specification
In a TSL specification of synchronization part, elements and its domains are restricted, as follows:

Element	Domain
fire	a finite set of gates except free gates
<gate name>	{empty,full,eos,m_1,m_2,...m_n,others}
<atomic object>:<state variable>	a domain of a state variable

A formula **fire(<gate name>)** corresponds to an action to fire the gate transition in MENDEL net. That is, "**fire(g)** is true" means "a gate transition **g** is fired". In the same way, "**<>fire(g)** is true" means "a gate transition **g** will be fired at some future time", and "**[]fire(g)** is true" means "a gate transition **g** is always fired".

Moreover, it is assumed that only one gate transition can be fired at a time, because of the single state condition for the element **fire**. Formulas **<gate name>(<state>)** and **<object>:<state variable>(<state>)** show states of gates and internal state variables respectively. That is, "**g1(full)** is true" means "a state of a gate **g1** is full", and so on. These formulas correspond to the <guard condition> in the <state transition rule> of a MENDEL synchronization part.

(3) Abbreviation
In a specification of synchronization parts, the following abbreviations are introduced for readability:

- a domain is uniquely determined by each element, therefore a domain can be omitted.
  ex. e(s,D) ---> e(s)
- fire(<gate name>) is abbreviated to <gate name>.
  ex. fire(g1) ---> g1
- if e∈E, s∈D, e(s) is abbreviated to e=s.
  ex. g1(full) --> g1=full, aaa:flag(on) --> aaa:flag=on.

(4) Example
A specification, "gates **g1** and **g2** are fired by turns unless the gate is full" is expressed by the following TSL formulas:

```
[]( g1 => @(~(g2=full) => g2 )) &
[]( g2 => @(~(g1=full) => g1 ))
```

7. SYNTHESIS OF SYNCHRONIZATION PART

The synchronization part of a MENDEL compound object can be synthesized from TSL specification and MENDEL net. TSL specification means constraints given by the programmer, and MENDEL net means constraints given by the structure of the body part. A valid synchronization part has to satisfy both of these constraints. The synchronization part synthesis consists of the following six steps (Fig.8).
(1) Synthesize a TSL model graph (TM) from a TSL specification (TS).
(2) Generate a MENDEL net (MN) from a body part of MENDEL program (BP).
(3) Extract a structure graph (SG) from a MENDEL net (MN).
(4) Generate a model graph (MG) by merging a TSL model graph (TM) and a structure graph (SG).
(5) Generate program codes of a synchronization part from a model graph (MG).

(1) TSL specification --> TSL Model Graph
TSL formulas can be translated into PTL formulas, as described in the section 6. PTL is decidable, and the tableau method [2,11] is one of decision procedures, which can compute all possible models of TSL formulas, represented in the form of a state transition graph. This is called TSL model graph (TM). In TM, each edge has a label that represents atomic propositions (<gate name> and <guard condition>) in conjunctive form. Here, every model of TSL formulas is represented as an infinite path on TM, which is identical with an infinite sequence of labels. A brief summary of the synthesis method is as follows:

(Step1-1) TSL formulas are translated into PTL formulas.
(Step1-2) PTL formulas are decomposed into current formulas and future formulas by the **decomposition procedure**. Current formulas does not include temporal operator. Future formulas are also decomposed into current and future formulas from the next time point of view. After every type (a finite number) of future formulas has been repeatedly decomposed, a

graph is derived. This graph is an incomplete model satisfying specifications other than the eventuality formulas, such as <>F, ~[]F and ~(~F1 $ F2).

(Step1-3) Edges with unsatisfiable eventuality formula are deleted from the graph by the **elimination procedure**. The graph remaining after the elimination procedure is a complete model of the initial TSL specification.

The TSL model graph for the TSL example in the previous section is shown in Fig. 9.

(2) MENDEL Body Part --> MENDEL Net
This step has been described in the section 5.

(3) MENDEL Net --> Structure Graph
A structure graph (SG) is a finite automaton over infinite transition sequences is generated by means of reduction in a reachability tree of MENDEL net (MN) (Appendix).

(4) TSL Model Graph and Structure Graph --> Model Graph
Both SG and TM are finite automata, but only TM has eventuality edges. Therefore, an intersection graph of SG and TM is first generated with inheriting eventuality edges from TM, and the **elimination procedure** is subsequently carried out for this graph involving eventuality edges, in the same way as in the tableau method in (1).

(5) Model Graph --> Synchronization Part
MG, which is also a finite automata, is nothing but a synchronization part of a MENDEL compound object. Transition rules are translated from MG. In atomic propositions on each label, an atomic proposition representing a gate corresponds to <gate name> in <transition rules> of a synchronization part, and the other atomic propositions corresponds to <guard condition> in <transition rules>. The transition rules are completed by adding the following "fairness strategy":

Fairness Strategy: If there are several enable transitions rules, one which has never been selected or for which the maximum time has elapsed from the last selection should be selected.

While the MENDEL interpreter fires a gate transition according to these transition rules and the fairness strategy, the sequence of fired gate is a model of the specification.

For clearer meanings of this synthesis, especially in the case when MG is empty, we define formal languages on a set of transitions and show a theorem without proof.

Definition
Let Σ be a set of transition. A infinite transition sequence is a infinite word on Σ. A language L(BP) denotes a set of all words which are possible on a body part BP. In the same manner, L(MN), L(SG), L(TM) and L(MG) denote sets of all words which are possible on MN, SG, TM, and MG respectively.

Corollary
(c1) $L(BP) \subset L(MN) \subset L(SG)$
(c2) $L(TS) = L(TM)$
(c3) $L(MG) = L(SG) \cap L(TM)$

Note: In this corollary, (c1) means MN is derived from BP with the loss of some constraints in the step (2), and SG is also derived with loss in the step (3). (c2) and (c3) give meanings of the synthesis step (1) and (5), respectively.

Theorem

If $L(MG) = \phi$, then $L(BP) \cap L(TS) = \phi$, which means that a TSL specification TS is unsatisfiable over a body part BP.

Note1: On this theorem, if MG is empty, the synthesis system notifies a programmer that TS is unsatisfiable over BP. In this case, the programmer has to modify TS or BP.

Note2: If MG is non-empty and the MENDEL interpreter executes according to the synchronization part derived from MG, there remains a possibility of the program falling into dead-lock or starvation, because $L(MG) \subset (L(BP) \cap L(TS))$ is not valid. This is due to the transformation from BP to SG with the loss of some constraints. However, we think this is small matter in the practical program synthesis.

8. SYNTHESIS EXAMPLE: HIERARCHICAL DINING PHILOSOPHERS

As an example of synthesis of MENDEL/88, let's consider the hierarchical dining philosophers. This program (#hierarchical_dining_philosopher) has two layers (Fig.10a). The top layer has three objects: two #phmain and one #logfile. Each #phmain forms the two dining philosophers problem. In #phmain, philosophers seize two forks and eat something, and then release the forks and think. In #logfile, "starttime" when one philosopher starts to eat and "finishtime" when he finishes eating are accumulated from each #phmain, and summarized last.

Here, we will show synchronization part synthesis only for #phmain. Other synchronization parts can be synthesized in the same manner. This subprogram consists of two atomic objects (#philosopher, #fork) and one compound object (#phmain) shown in Fig.10b. MENDEL net (MN) of the body part (BP) of #phmain are given by Fig.10c. This MENDEL net presents the essential property of the program. A structured graph (SG) is generated from MN. To synthesize a specification part of #phmain, we give a simple TSL specification TS:

```
TS:   [ ](r11 <=> @r12) & [ ](r21 <=> @r22) & ---  (1)
      [ ]<>(s11 # s12) & [ ]<>(s21 # s22))      ---  (2)
```

This specification means: (1) The philosopher must continuously release both the forks, which prohibits the philosopher from keeping one of forks constantly. (2) Both of philosophers are dead-lock free and starvation free. Note that since t11, t12, t21, t22, f1, and f2 are free gates, they do not appear in TS and SG. From TS and SG, a model graph MG (Fig.10d) and the following specification part are synthesized:

```
sync:{ ( [    trans(n1,n2,[],[s21]),  trans(n1,n3,[],s22),
              trans(n1,n4,[],[s11]),  trans(n1,n5,[],s12),
              trans(n2,n6,[],[s22]),  trans(n3,n6,[],s21),
              trans(n4,n8,[],[s12]),  trans(n5,n8,[],s11),
              trans(n6,n9,[],[r21]),  trans(n9,n11,[],r22),
              trans(n8,n10,[],[r11]),  trans(n10,n11,[],r12),
              tarns(n11,n2,[],[s21]),  tarns(n11,n3,[],s22),
              tarns(n11,n4,[],[s11]),  tarns(n11,n5,[],s12)    ],
              n1 )} ;
```

9. Related Works

Our synthesis method is the extension of Manna & Wolper's work [1,2]. In Manna & Wolper, the only synchronization part of CSP is synthesized, while our method can synthesize both synchronization and body parts consistently using Petri net. Moreover, in their method, there exists one scheduler process, with which each process can only communicate, that is, normal processes can not communicate with each other. This scheduler process is indispensable for the synthesis using temporal logic, however, seems strange for CSP programmers. In our concurrent program model and MENDEL/88, this scheduler is well formalized as a transition control program on Petri net.

Some works are carried out for synthesis and verification using Petri net and temporal logic [12,13]. These works are similar to our approach in the section 7, but differ in correspondence between Petri net and temporal logic. In these works, atomic propositions correspond to marking in places, while atomic propositions correspond to transition firing in our method.

ENVISAGER system [14] is a visual programming environment on UNIX workstation, which is similar to MENDELS ZONE. This system adopts Interval Temporal Logic as a specification language, however, Interval Temporal Logic is used only for simulation, and not for program synthesis.

10. CONCLUSION

We have proposed a hierarchical concurrent programming language, MENDEL/88. For this language, the program synthesis method is presented. This method consists of two major steps: (1) Construction of the body part by reusable objects (2) Synthesis of the synchronization part which is consistent with the body part, from a TSL specification and a MENDEL net. Unique features of this method include: (1) A target language MENDEL/88 involving a unique hierarchical synchronization mechanism using gates, and (2) A combination of software reuse and synthesis using temporal logic and Petri net. We believe this approach will be practical enough to help prototyping on a qualified domain, such as concurrent business transactions. This is the reason why data flow diagrams written by the Real-Time SA are basically easy to be converted to MENDEL programs.

At present, much remains to be explored:
(1) Main drawback of synthesis using Petri net and temporal logic is the exponential time complexity. The hierarchical synthesis approach will be able to make up for this drawback.
(2) The degree of concurrency is low because of the limitation wherein only one gate transition can be fired at a time, except for "free gates". It seems possible to relax this limitation for some independent gates in the future.

ACKNOWLEDGMENT

This research has been supported by ICOT. We would like to thank Ryuuzou Hasegawa of ICOT for their encouragement and support. We are also grateful to Seiichi Nishijima, Takeshi Kohno, and Hideo Nakamura of Systems & Software Engineering Laboratory, TOSHIBA Corporation, for providing continuous support, and other members who are developing MENDELS ZONE. Many thanks are due to Mutumi Fujihara of TOSHIBA Corp for variable suggestions for designing MENDEL/88.

REFERENCES

[1] Wolper, P., Synthesis of communicating processes from temporal logic specification, STAN-CS-82-925, Stanford University, 1982.
[2] Manna, Z. and Wolper, P., Synthesis of communicating processes from temporal logic specification, ACM Trans. on Programming Languages and Systems, Vol.6, No.1, pages 68-93, 1984.
[3] Clarke, E. M. and Emerson, E. A., Design and synthesis of synchronization skeletons using branching time temporal logic, Logics of programs (Proceedings 1981), Lecture Notes in Computer Science (LNCS) 131, Springer-Verlag, pages 52-71, 1982.
[4] Fujita, M., Tanaka, H.,and Moto-oka, T., Specifying hardware in temporal logic & efficient synthesis of state diagrams using Prolog, Proc. of FGCS'84, 1984.
[5] Honiden, S., Uchihira, N., and Kasuya, T., MENDEL: PROLOG BASED CONCURRENT OBJECT ORIENTED LANGUAGE, Proc. of COMPCON'86, pages 230-234,1986.
[6] Ward,P.,T.,The transformation schema: An extension of the data flow diagram to represent control and timing, IEEE Trans. on SE, vol.SE-12, No.2, 1986.
[7] Uchihira, T., Kasuya, T., Matsumoto,K., and Honiden, S., Concurrent Program Synthesis with Reusable Components Using Temporal Logic, Proc of COMPSAC87, 1987.
[8] Campbell,R.H. and Habermann,A.N., The Specification of Process Synchronization by Path Expression, LNCS16, Springer-Verlag, 1974.
[9] Hoare,C.A.R., Communicating Sequential Processes, CACM Vol.21,No.8, 1978.
[10] Ueda,K., Guarded Horn Clauses, Technical Report TR-103, ICOT, 1985.
[11] Plaisted, D. A., A Decision Procedure for Combinations of Propositional Temporal Logic and Other Specialized Theories, Journal of Automated Reasoning 2, pages171-190, 1986.
[12] Tuominen,h.,Temporal logic as a query language for Petri net reachability graphs, 7th European Workshop on Application and Theory of Petri nets, 1986.
[13] Katai, O. and Iwai, S., Construction of Scheduling Rules for Asynchronous, Concurrent Systems Based on Tense Logic (in Japanese), Trans. of SICE (Japan) vol.18 no.12, 1982.
[14] Gonzalez,J.P., Urban,J.E., ENVISAGER: A Visual, Object-Oriented Specification Environment for Real-Time Systems, Proc of 4th International Workshop on Software Specification and Design, 1987.

APPENDIX

A Structure Graph Generation Algorithm

(Step1) Make a reachability tree (RT) from MENDEL net (MN).

(Step2) If MN is bounded, RT can easily be transformed to an equivalent state transition graph STG. If MN is unbounded, generate STG by relaxing the RT with the threshold of token number.

(Step3) Reduce STG to the structure graph (SG) by eliminating method transitions, which are regarded as the ε-action in the automaton.

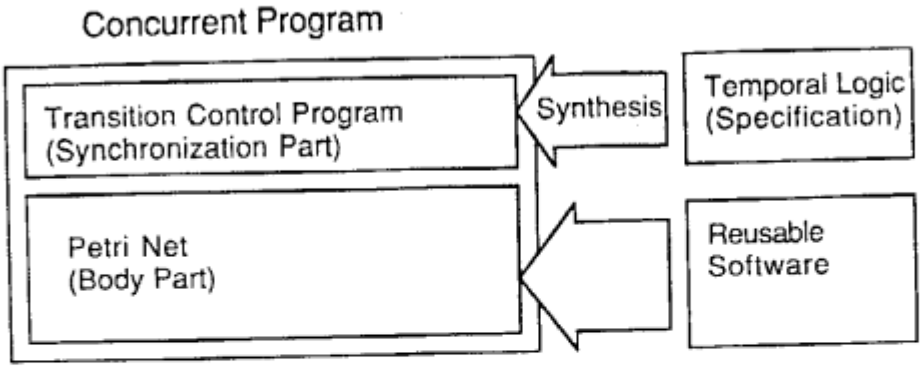


Fig.1. Relation of Petri Net and Temporal Logic on Concurrent Program Model

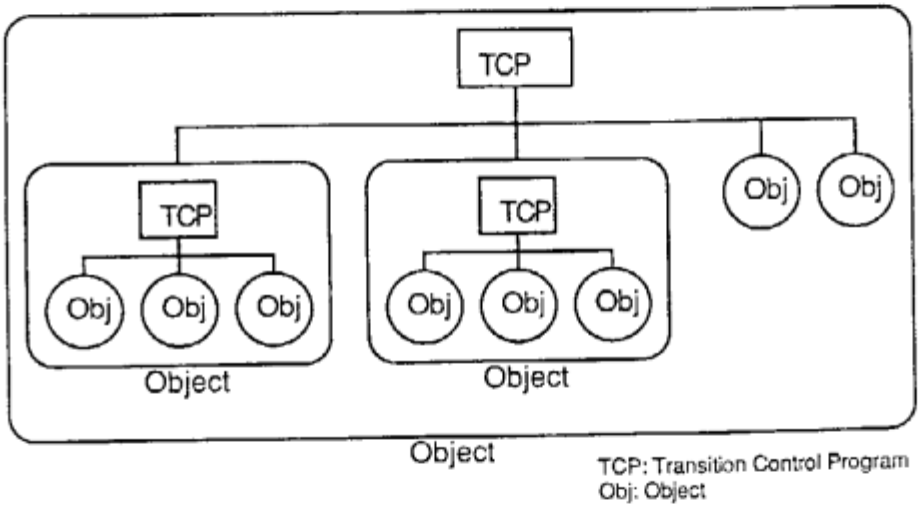


Fig.2. Hierarchical Structure of Concurrent Program

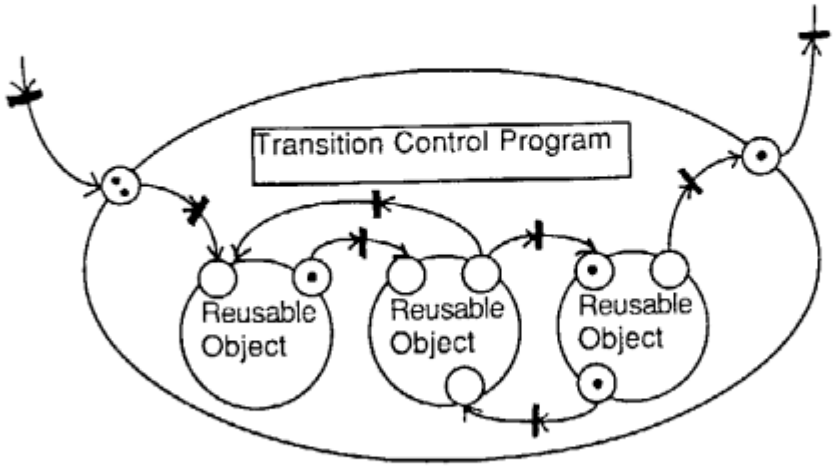


Fig.3. Reusable Object and Transition Control Program

```
%   ************
%   * Factrial *
%   ************

%*** GENERATOR ***

atomic object generator:{
    dec:{
            inport(number) ;
            outport(send,end) ;
            state(s:[active,sleep]!sleep) ;
            data(count!0)
    } ;
    meth:{
            method(s?sleep,number?N,s!active,count!N) ;
            method(s?active,count?N,count!M,send!N) <-
                N > 0 | M is N - 1 ;
            method(s?active,count?N,s!sleep,end!signal) <-
                N = 0 | true ;
    };
    junc:{ }
}.

%*** CALCULATOR ***

atomic object calculator :{
    dec:{
            inport(receive,finish) ;
            outport(answer) ;
            data(fact!1)
    } ;
    meth:{
            method(receive?N,fact?M,fact!K) <-
                true | K is N*M ;
            method(finish?signal,fact?M,answer!M,fact!1) ;
    } ;
    junc:{}
}.

%*** Factorial ***

object factrial :{
    dec:{
            inport(number) ;
            outport(answer) ;
            mgate(g1,g2,g3) ;
            sgate(s1,s2)
    } ;
    body:{
            innode(number?[g1]) ;
            outnode(answer![g3]) ;
            generator(number![g1],send?[g2],end?[s1]) ;
            calculator(receive![g2],end![s2],answer?[g3])
    } ;
    sync:{
          ([trans(n0,n1,[],g1),
            trans(n1,n1,[],g2),
            trans(n1,n2,[g2=empty],s1),
            trans(n2,n3,[],s2),
            trans(n3,n0,[],g3)],
            n0)
    }
}.
```

Fig.4. MENDEL/88 Program Example (Factorial Program)

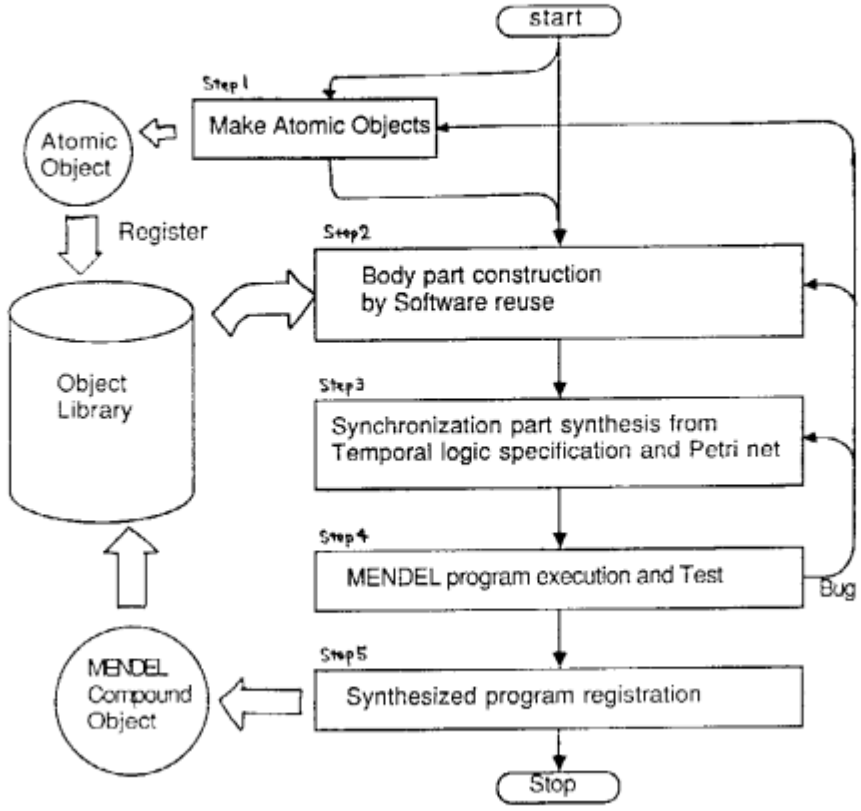


Fig.5. MENDEL Program Synthesis in MENDELS ZONE

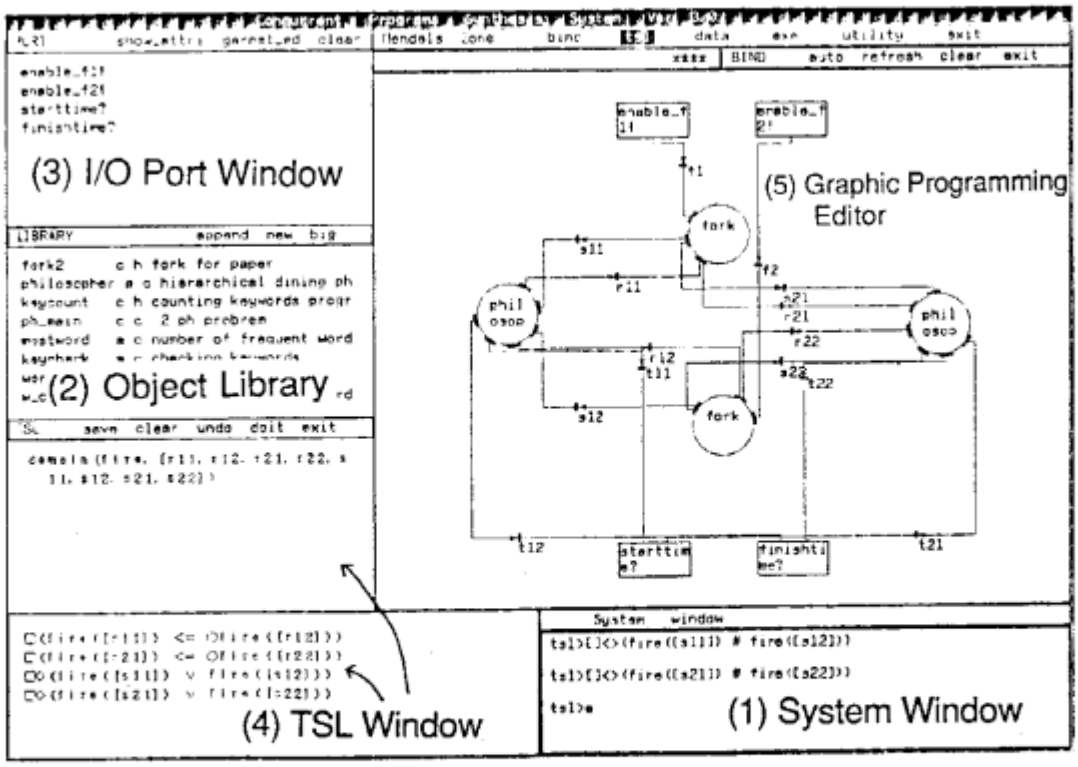


Fig.6. User Interface of MENDELS ZONE

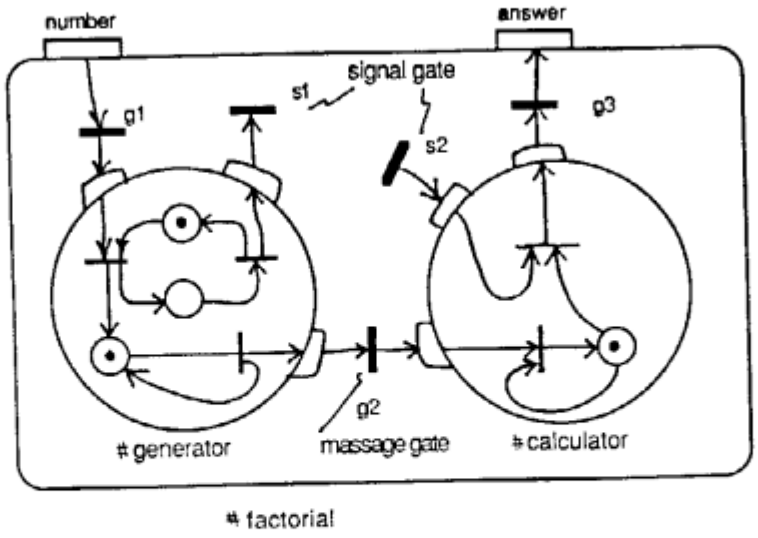


Fig.7. MENDEL Net for Factorial Program

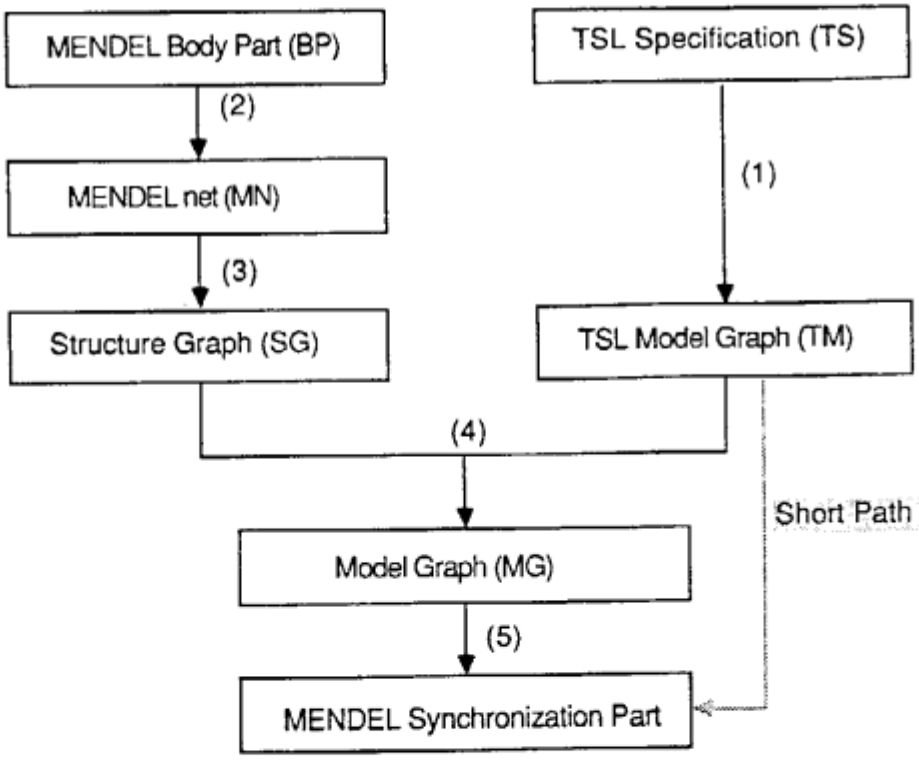


Fig.8. Overview of Synchronization Part Synthesis

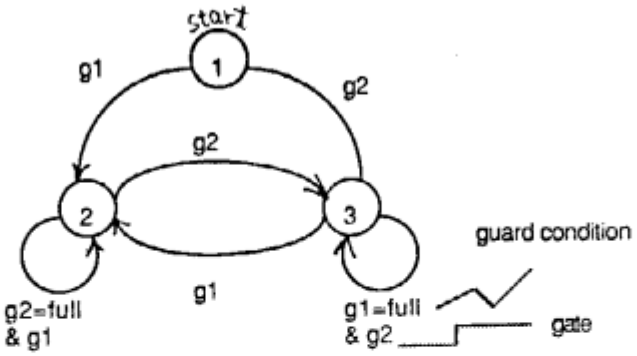
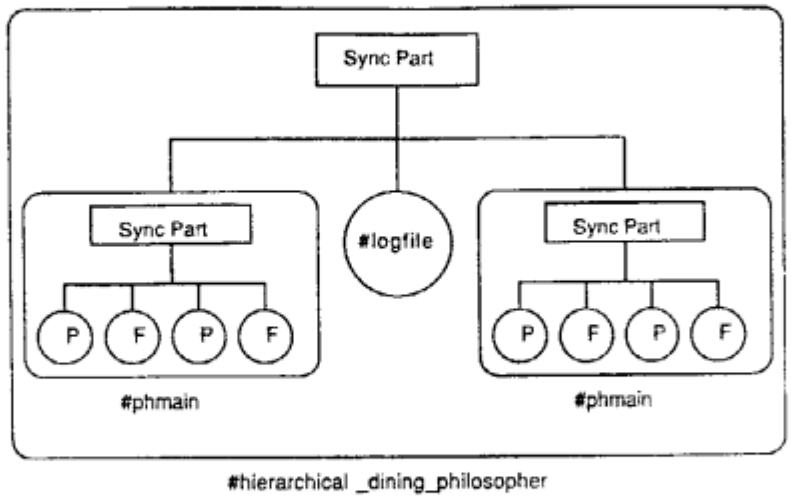
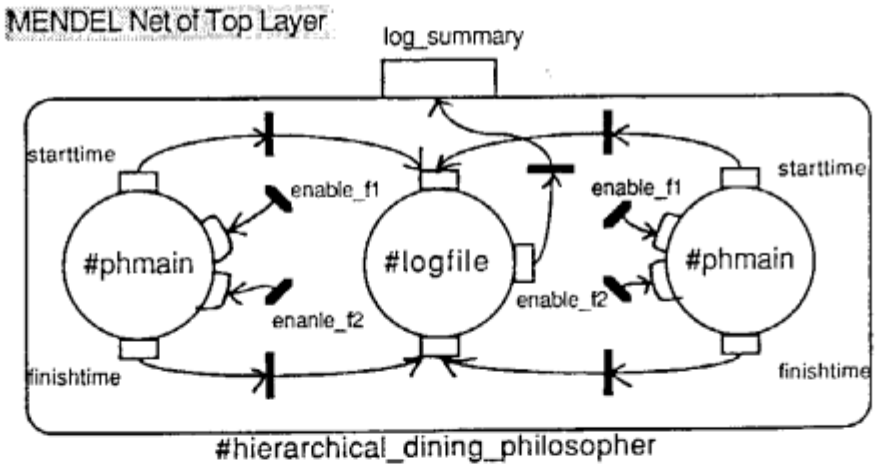


Fig.9. Example of TSL Model Graph



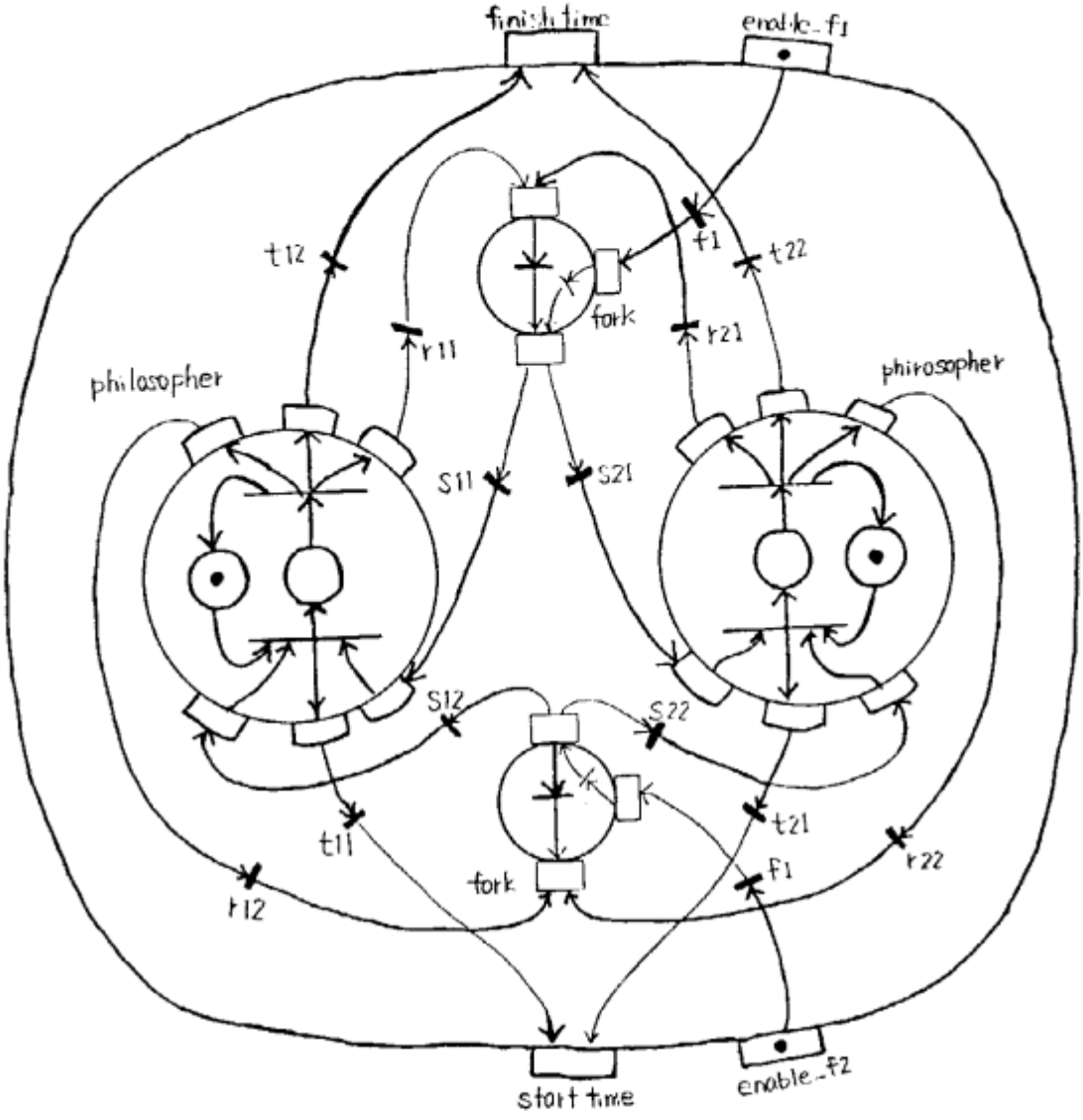


(a) Hierarchical Structure

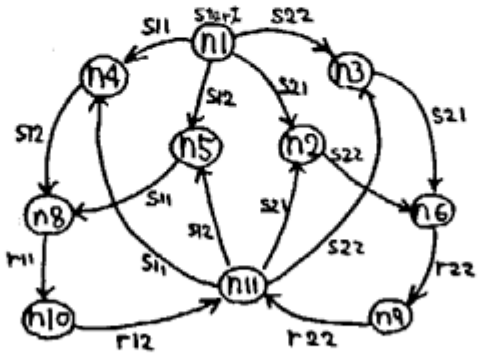
Fig.10. Example: Hierarchical Dining Philosophers

```
%   ************************************
%   * Hierarchical Dining Philosophers *
%   ************************************

%*** MAIN PROGRAM (#phmain) ***

object phmain :[
    dec:[
            inport(enable_f1,enable_f2) ;
            outport(starttime,finishtime) ;
            mgate(s11,s12,s21,s22,r11,r12,r21,r22) ;
            fgate(f1,f2,t11,t12,t21,t22)
    ] ;
    body:[
            innode(enable_f1?[f1],enable_f2?[f2]) ;
            outnode(starttime![t11,t21],finishtime![t12,t22]) ;
            philosopher(inrfork![s11],inlfork![s12],outrfork?[r11],outlfork?[r12],
                        starttime?[t11],finishtime?[t12]) ;
            philosopher(inrfork![s21],inlfork![s22],outrfork?[r21],outlfork?[r22],
                        starttime?[t21],finishtime?[t22]) ;
            fork(enable![f1],return![r11,r21],rent?[s11,s21]) ;
            fork(enable![f2],return![r12,r22],rent?[s12,s22])
    ] ;
    sync:[
             %*** Noy Yet Synthesized ***
    ]
].

%*** PHILOSOPHER (#philosopher) ***

atomic object philosopher :[
    dec:[
            inport(inrfork,inlfork) ;
            outport(outrfork,outlfork,starttime,finishtime) ;
            state(s:[thinking,eating]!thinking)
    ] ;
    meth:[
            method(s?thinking,inrfork?ok,inlfork?ok,s!eating,starttime!T)
                <- true | get_time(T) ;
            method(s?eating,outrfork!free,outlfork!free,s!thinking,finishtime!T)
                <- true | get_time(T) ;
    ] ;
    junc:[
            get_time(T) :- T is cputime ;
    ]
].

%*** FORK (#fork) ***

atomic object fork :[
    dec:[
            inport(return,enable) ;
            outport(rent)
    ] ;
    meth:[
            method(enable?signal,rent!ok) ;
            method(return?free,rent!ok) ;
    ] ;
    junc:[]
].
```

(b) MENDEL/88 Program of #phmain



(c) MENDEL Net of #phmain



(d) Model Graph of #phmain