ソフトウェア工学 69－7
（1989. 11. 24）

階層的ペトリネットによる並行プログラムの設計法
――ペトリネットと時相論理を用いた検証と合成――

内平直志　　本位田真一

東芝　システム・ソフトウェア技術研究所

ペトリネットと時相論理は，ともに並行システムの仕様記述手段として有効であり，それぞれ異なる観点からの記述ができる．すなわち，ペトリネットはシステムの動的行動を操作的に記述でき，時相論理はシステムの特性や制約を宣言的に記述できる．設計者は，全ての仕様を宣言的に記述できるわけではないし，逆に全ての制約条件を操作的に記述するならそれはもはや詳細設計である．操作的にも宣言的にも仕様が記述できる枠組みが現実的に有効である．そこで，ペトリネットと時相論理の融合は，並行プログラムの仕様記述言語として非常に有望なアプローチであるといえる．本報告では，まずペトリネットと時相論理を融合したクラスを示し，それを用いた並行プログラムの検証と合成手法を示す．

Verification and Synthesis of Concurrent Programs Using Petri Nets and Temporal Logic

(Preliminary Report)

Naoshi Uchihira　　Shinichi Honiden

Systems & Software Engineering Laboratory
TOSHIBA Corporation
Yanagi-cho 70, Saiwai-ku, Kawasaki, Kanagawa 210, JAPAN
uchi%ssel.toshiba.junet@uunet.uu.net

ABSTRACT

Both Petri net and temporal logic have been widely used to specify concurrent systems. Petri net is appropriate to explicitly specify the behavioral structures of systems, while temporal logic is appropriate to specify the properties and constraints of systems. Since one can complement the other, using a combination of Petri net and temporal logic is a highly promising approach to analyze, verify and synthesize concurrent programs. Several reports on research efforts [CK87,SL89,HR89] have been presented to combine a non-restricted Petri net with propositional temporal logic. However, the Petri net combined with temporal logic in these reports is so powerful that it is inappropriate for use in automatic program verification and synthesis, because of its undecidability. This paper reports a class that is decidable and shows how to verify and synthesize concurrent programs using Petri nets and temporal logic. Especially, in the proposed method, a concurrent program is compositionally synthesized with reusable components.

1. INTRODUCTION

The Petri net is widely accepted as a graphical and mathematical modeling tool, applicable to concurrent systems [Pe81]. Temporal logic is also successfully applied as a tool for the verification [Pn77] and synthesis [MW84] of concurrent systems. The Petri net and temporal logic have different features with each other. The Petri net is suited for modeling the behavioral structures of a concurrent system, and temporal logic is suited for specifying the timing constraints of a system. In other words, one can model or program concurrent systems operationally using Petri nets, while one can specify them declaratively using temporal logic. For example, a prohibiting constraint, such as "once an error event occurs, a start event must not be activated" can be described explicitly by temporal logic, but it can only be described implicitly by Petri nets. Reversely, it is often tedious work to describe concrete action sequences by temporal logic, which can easily be accomplished by Petri nets. It is an excellent idea to combine the Petri net and temporal logic as a specification language for analyzing, verifying and synthesizing concurrent systems, because the Petri net and temporal logic complement each other. However, most classes [CK87,SL89,HR89], in which the unbounded Petri net is combined with linear time propositional temporal logic, are undecidable in regard to the emptiness (satisfiability) problem. In this paper, we select a class that is decidable in Section 3 and show verification and synthesis methods for concurrent programs using Petri nets and temporal logic, in Sections 4 and 5, respectively.

2. PRELIMINARIES

Definition 1 (Petri net) [Mu89]:
A Petri net is a 5-tuple, $N=(P,T,F,w,M_0)$ where:
$P=\{p1,p2,...,p_m\}$ is a finite set of places,
$T=\{t1,t2,...,tn\}$ is a finite set of transitions,
$F \subset (P\times T)\cap(T\times P)$ is a set of arcs (flow relation),
$w: F\rightarrow\{1,2,3,...\}$ is a weight function,
$M_0: P\rightarrow\{0,1,2,...\}$ is the initial marking,
$P\cap T=\emptyset$ and $P\cup T\neq\emptyset$.
A marking in a Petri net is changed according to the following firing rule:
1) A transition is said to be *enabled*, if each input place p of t is marked with at least w(p,t) tokens, where w(p,t) is the weight of the arc from p to t.
2) Only one of the enabled transitions can fire at a time.
3) Firing of an enabled transition t removes w(p,t) tokens from each input place p of t, and adds w(t,p) tokens to each output place p of t, where w(t,p) is the weight of the arc from t to p.

Definition 2 (Labelled Petri net)
A labelled Petri net is a 6-tuple, N = (P, T, F, w, h, M0) where:
(P,T,F,w,M_0) is a Petri net, and
$h: T\rightarrow\Sigma$ (alphabet) $\cup$ {λ (empty string)} is a labelling function.

Let X (Σ) be a finite set (alphabet). ω means infinitely many. The set of all finite sequences, including an empty sequence, over X is denoted by X^*, and the set of all infinite sequences over X is denoted by X^ω. $X^\infty = X^* \cup X^\omega$. The labelling function $h:T\rightarrow\Sigma$ is extended to $h:T^\infty\rightarrow\Sigma^\infty$ by $h(\theta)(i) = h(\theta(i))$ for all $\theta\in T^\infty$ and $1\leq i\leq|\theta|$, where $x(i)$ means the i-th element in sequence x and $|\theta|$ is the length of θ. A sequence of transition ($\theta \in T^*$) is called a firing sequence for the Petri net $N=(P,T,F,w,M_0)$, if the successively firing sequence of the transitions is allowed by the firing rule in N; an infinite sequence of the transitions ($\theta \in T^\omega$) is an infinite firing sequence if every prefix is a firing sequence. The set of all (infinite) firing sequences of N is denoted by F(N) ($F_\omega(N)$).

Definition 3 (Petri net language)
L(N) is a Petri net language generated from Petri net N if $L(N) \equiv \{h(\theta) \mid \theta\in F(N)\}$, and $L_\omega(N)$ is a Petri net ω language generated from Petri net N, if $L_\omega(N) \equiv \{h(\theta) \mid \theta\in F_\omega(N)\}$

Definition 4 (Linear time propositional temporal logic) :
(1) Syntax
Linear time propositional temporal logic (LPTL) formulas are built from
- A set Prop of atomic propositions: p1,p2,p3,...
- Boolean connectives: $\wedge$, $\neg$
- Temporal operators: X ("next"), U ("until")

The formation rules are:
- An atomic proposition $p \in$ Prop is a formula.
- If f1 and f2 are formulas, so are $f1\wedge f2$, $\neg f1$, Xf1, f1 U f2.

(2) Semantics
The operators intuitively have the following meanings:
$\neg$: NOT, $\wedge$: AND, Xf (read next f) : f is true for the next state, f1 U f2 (read f1 until f2) : f1 is true until f2 becomes true. The precise semantics are given as a Kripke structure in [Wo89].

We use Ff ("eventually f") as an abbreviation for (true U f) and Gf ("always f") as an abbreviation for $\neg F\neg f$. Also, $f1 \vee f2$ and $f1 \supset f2$ abbreviate $\neg(\neg f1 \wedge \neg f2)$ and $\neg f1 \vee f2$, respectively.

Definition 5 (Büchi sequential automaton) [Wo89]:
Büchi sequential automaton is a tuple $A=(\Sigma,S,\rho,s_0,F)$, where
- Σ is an alphabet,

- S is a set of states,
- $\rho: S \times \Sigma \rightarrow 2^S$ is a nondeterministic transition function,
- $s0 \in S$ is an initial state, and
- $F \subset S$ is a set of designated states.

A run of A over an infinite word w=t1t2..., is a sequence s0,s1,..., where $s_i \in \rho(s_{i-1}, t_i)$, for all $i \geq 1$. A run s0,s1,... is accepting if for some $s \in F$ there are infinitely many i's such that si=s. An infinite word θ is accepted by A if there is an accepting run of A over θ. The set of all words accepted by A is denoted L(A).

Theorem 1:
Given an LPTL formula f, one can build a Büchi sequential automaton $A_f=(\Sigma,S,\rho,s0,F)$, where $\Sigma=2^{Prop}$, such that L(Af) is exactly the set of sequences satisfying formula f.
Proof. Cf. [WVS83]. ■

Definition 6 (Single Event Condition) [MW84]:
A single event condition is

$$A\,((\bigvee_{1 \leq i \leq n} pi) \wedge (\bigwedge_{1 \leq i \leq j \leq n} \neg(pi \wedge pj)),$$

where p1,...,pn are all atomic propositions.

A single event condition provides that just only one atomic proposition is true at any moment. When we build a Büchi sequential automaton Af '=(Σ,S,r,s0,F), where f ' is f with a single event condition, we can make Σ=Prop in place of $\Sigma=2^{Prop}$, because only one atomic proposition is true.

Definition 7:
Ls(f) is a ω language generated from an LPTL formula f with a single event condition if Ls(f)=L(Af ') where f ' is f with a single event condition and Σ of Af ' is Prop.

Lemma 1:
Given an LPTL formula f, $Ls(f)^c = Ls(\neg f)$, where $Ls(f)^c = \Sigma^\omega - Ls(f)$.
Proof.
In [WVS83], it is proved that $L(A_{f1})^c = L(A_{f2})$, where f2 = ¬ f1 and no single event condition is assumed. This lemma is a special case of the theorem. ■

3. HOW TO COMBINE A PETRI NET AND TEMPORAL LOGIC

There are several ways to combine a Petri net with temporal logic. The key point in combining is what the atomic proposition in temporal logic corresponds to in the Petri net. Some correspondences will be shown between atomic propositions in LPTL and Petri net properties:
a) atomic proposition p is true iff place p has at least one token.
b) atomic proposition ge(p,c) is true iff place p has at least c tokens.
a) is a special case of b), i.e. ge(p,1).
c) atomic proposition en(t) is true iff transition t is enabled.
d) atomic proposition fi(t) is true iff transition t fires.

Here, fi(t) ⊃en(t) always holds.

Paper	Type	Petri net	Emptiness Problem
[KI82]	a	safe	decidable
[CK87]	b,c,d	normal	undecidable
[SL89]	a,c,d	normal	undecidable
[HR89]	b,c,d	conflict-free	undecidable
[UKH89]	d	bounded	decidable

Table 1 Several Combination of Petri Net and LPTL

For these correspondences, several research results are presented as shown in Table 1. It can be seen that the emptiness problem becomes undecidable in some Petri nets combined with temporal logic. Some are decidable but are restricted to bounded ones. Our purpose is to select an unbounded Petri net class combined with temporal logic in which the emptiness problem is decidable. The reason is that decidability is necessary for automatic program verification and synthesis, and unboundedness of the Petri net is necessary for modeling asynchronous communication in concurrent programs.

Here, we adopt only d-type correspondence and combine the Petri net and LPTL in the world of formal language over a set of transitions. In a previous section, it was pointed out that Petri net language L(N) is generated from Petri net N, and ω language L(f), which is exactly the set of sequences satisfying the LPTL formula f, can be represented as L(Af) where Af is a Büchi sequential automaton.

When combining Petri net N and temporal logic f, all transitions of N do not necessarily correspond to atomic propositions. Some transitions may be invisible to a user who describes temporal logic specifications. Let T be a set of all transitions of N and $\Sigma \subset T$ is a set of visible transitions. A labelling function h: $T \rightarrow \Sigma$ is defined such that h(t)=t, if $t \in T$ is visible, and h(t)=λ, if $t \in T$ is invisible. We now define a new formal language from L(N) and Ls(f).

Definition 8:
Let T be a set of all transitions, $\Sigma \subset T$ be a set of visible transitions, and $Prop \subset \Sigma$ be a set of visible transitions, which appear in a temporal logic formula f. $L(N,f) \equiv L\omega(N) \cap Ls(f)$ where Lω(N) is a language generated from Petri net N=(P,T,w,h,M0), h: $T \rightarrow \Sigma$ and Ls(f) is a

language generated from f under a single event condition.

Theorem 2:
For a given labelled Petri net N=(P,T,w,h,M0) and LPTL formula f composed of a set of atomic propositions Prop, the emptiness problem of L(N,f) is decidable.

Proof.
It is sufficient to prove that the emptiness problem $L\omega(N) \cap Ls(f)$ is decidable. To begin with, a procedure is provided which constructs a coverability graph CG with A_f:

Main Algorithm
1) Make a Büchi sequential automaton $A_f=(Prop,S,\rho,s_0,F)$ accepting Ls(f).
2) Expand Af into Af ' = (T,S',ρ',s_0,F) appending dummy states and transitions for invisible transitions (T-Σ) according to [UKS89].
3) Construct a coverability graph CG. CG is a labelled directed graph. Each node x in CG is represented as a vector x = (x1,x2,...,xk,s,flag) where $x_i \in \{0,1,...\} \cup \{\omega\}$ $(1 \le i \le k)$, |P|=k, $s \in S'$, flag∈ { d (designated node), n (normal node) }. Each edge e=(xi,xj) in CG is labelled with an element of T. CG is constructed as follows:
2-1) Start with a graph CG containing one initial node $x^0=(x^0_1,...,x^0_k,s_0,flag)$ where $(x^0_1,...,x^0_k)$ is an initial marking M_0, $s_0 \in S'$ is an initial state of A_f', flag=n(normal).
2-2) Repeatedly apply steps 2-3) to the CG nodes until they have been applied to all nodes.
2-3) Let x be an unapplied node in CG with x=(x1,...,xk,s,flag).
(Case 1) If there is no enabled transition $t \in \Sigma$ on x, x has no child node; otherwise,
(Case 2) For every enabled transition $t \in \Sigma$ on x and every $s' \in \rho(s,t)$, add new child edges and/or nodes to CG by the following Graph Addition Procedure.

Graph Addition Procedure
For a given x, $t \in \Sigma$ and $s,s' \in S'$, add new nodes and edges to CG according to 1) - 4).
1) Make a new vector x'=(x'1,x'2,...,s',flag), such that flag=d $s' \in F$, otherwise flag=n, and each xi' $(1 \le i \le k)$ is computed as follows:
(Case 1) if $x_i = \omega$, $x'_i = \omega$ $(1 \le i \le k)$;
(Case 2) if there is an ancestor y=(y1,...,yk,s',$flag_y$) of x such that $y_j \le x_j$ - w(pj,t) + w(t,pj) for all j $(1 \le j \le k)$ and $y_i < x_i$ - w(pi,t) + w(t,pi) for some i $(1 \le i \le k)$, then x'i=ω; otherwise,
(Case 3) x'i = xi - w(pi,t) + w(t,pi).
2) If x' is new in CG, add a new node x' and a new edge e=e(x,x') labelled with t, otherwise add only a new edge e=(x,x') labelled t.

The above algorithm always terminates, because the number of nodes and edges is finite.
Claim: $L\omega(N) \cap Ls(f) \neq \emptyset$ iff there exists a cycle $c=x_0x_1...x_kx_0$ such that x_0 is a designated node and $\Delta(c) \ge 0$, where Δ(c) is a difference vector of the number of tokens in each places between a start point and an end point in c.
The above claim follows directly from the result of [SCFM84]. Furthermore, it is decidable if there exists such a cycle from [VJ85]. ■

When L(N,f) is nonempty, it is very important to find a concrete sequence accepted by L(N,f). for the sake of program synthesis.

Definition 9 :
A linear Σ-labeled graph is a tuple $G=(V,v_0,h,\rho)$, where V is a set of nodes, v_0 is an initial node, and $\rho: V \to V$ is a transition function and $h: V \to \Sigma$ is a labelling function.

Note that a linear Σ-labeled graph defines one deterministic sequence obtained by infinite unwinding from node v_0.

Theorem 3:
Given L(N,f) that is nonempty, we can construct a linear T-labeled graph G=(S,s0,ρ,h) defining a concrete firing sequence $\theta \in L(N,f)$.
Proof. Cf. [VJ85] ■

4. CONCURRENT PROGRAM VERIFICATION

Consider concurrent program verification focusing on the behavioral properties. After retracting the basic behavioral structures represented by Petri nets from concurrent programs, it is possible to analyze the behavioral properties of programs. This verification means to check whether or not a given Petri net satisfies a given specification. What language should be used to describe the specifications? Temporal logic was adopted where atomic propositions correspond to transition firing as described in the previous section. However, only the ω Petri net language $L_\omega(N)$ is considered there, which doesn't care for finite behaviors of N including deadlocks. Therefore, Petri net N is extended to Petri net N_ω which is made deadlock-free by adding a dummy transition **nop** (no operation) in Fig.1.

Fig.1 nop

It will be shown how to verify that a given concurrent program meets a given specification. A concurrent program is represented as a Petri net N_ω, and a specification is described as a temporal logic formula f. Thus, to verify that the program meets the specifications, it suffices to check

$Ls(f) \supset L_\omega(N_\omega)$, that means each of all possible computations in Petri net N_ω is a model of temporal logic formula f.

Definition 10:
A deadlock-free Petri net N_ω satisfies the temporal logic specification f with a single event condition iff $Ls(f) \supset L_\omega(N_\omega)$. And it is called the verification problem to decide whether N_ω satisfies f.

Theorem 4:
The verification problem is decidable.
Proof. From lemma 1, $Ls(f) \supset L_\omega(N_\omega) \equiv Ls(\neg f) \cap L_\omega(N_\omega) = \emptyset$. It is decidable from Theorem 3. ■

It will now be made clear what the inputs and outputs are:

INPUT:
Concurrent program structure
(represented by Petri net N_ω),
INPUT:
Specification
(represented by temporal logic f),
OUTPUT: Yes/no,

where "yes" means that the program satisfies the specification, and "no" means otherwise.

Some examples will be shown to clear what is possible and what is impossible in this verification method:

<Possible to verify>
1) Mutual exclusion
ex. Intervals [t1,t2] and [t3,t4] between two transitions do not overlap each other:
$G(t1 \supset X[\neg(t3 \vee t1) U t2) \wedge G(t3 \supset X[\neg(t1 \vee t3) U t4)$
2) Partial ordering among transition firing
ex. Transition t1 and t2 fire in turn:
$G(t1 \supset X[\neg t1 U t2) \wedge G(t2 \supset X[\neg t2 U t1)$
3) Firing prohibition
ex. $G(t1 \supset XG \neg t2)$
4) Deadlock inevitability
ex. $FG \neg t$

Note: Each place in a Petri net represents either an internal state or a communication buffer in a concurrent program. Places representing states can be taken place of intervals of two transitions [t1,t2].

<Impossible to verify>
1) Number of tokens
It is impossible to verify about the number of tokens in the places, which is used to represent reachability and boundedness properties.
2) Possibility of deadlock (liveness property)
This arises from the introduction of nop.

The blind side of this verification can be complemented by the traditional analysis method for Petri nets.

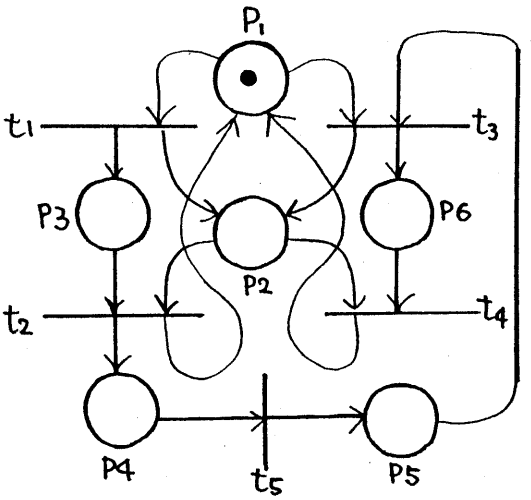
Example of Verification:

As a simple example, verifying a concurrent program, let's consider a mutual exclusion problem containing unbounded buffers. Note that a bounded Petri net that is equivalent to a finite state program is easier to verify using Clarke's model checker [CES86]. A target program is illustrated in Fig.2, where places P4 and P5 are unbounded buffers. And specification f is given that intervals [t1,t2] and [t3,t4] satisfy a mutual exclusion condition as follows:

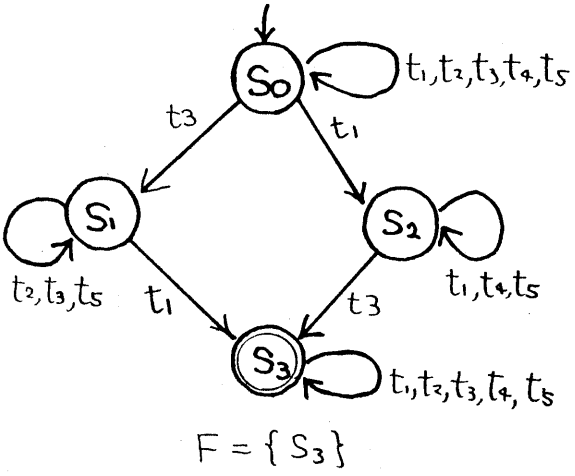
Specification f:

$f \equiv G(t1 \supset X[\neg t3 U t2)) \wedge G(t3 \supset X[\neg t1 U t4))$
$\neg f \equiv F(t1 \wedge X\neg(\neg t3 U t2)) \vee F(t3 \wedge X\neg(\neg t1 U t4))$

Deadlock-free Petri net N_ω:



Buchi Sequential Automaton $A_{\neg f}$:



Reachability Graph CG:

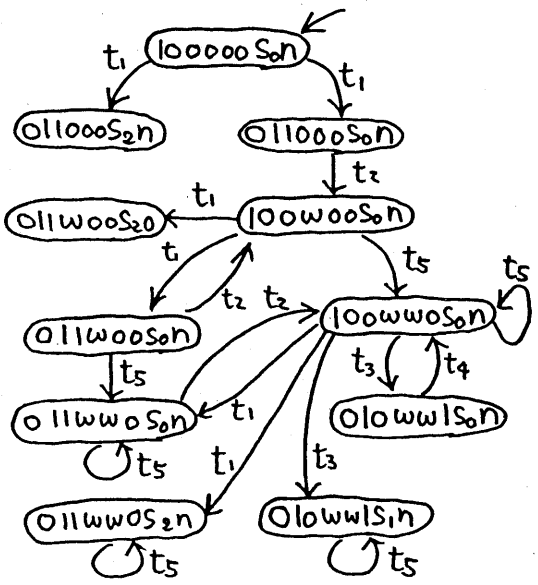


Fig.2 Verification example

In CG, there exists no cycle including a designated node, that means $Ls(\neg f) \cap L_\omega(N_\omega) = \emptyset$ from Theorem 2. We conclude N_ω satisfies f.

5. COMPOSITIONAL SYNTHESIS WITH REUSABLE COMPONENTS

This section describes a method to synthesize a concurrent program with reusable components by program tuning. The goal programs are synthesized by tuning up reused programs that are represented by Petri nets to satisfy the given specification. This method differs from other synthesis methods [MW84,CE82] that also use the temporal logic specification, in the point of utilizing software reuse [BP84,KG87,UKMH87]. The model building technique in Theorem 3 is used in this tuning process.

At first a flat synthesis method is explained, and then a compositional synthesis method is mentioned briefly.

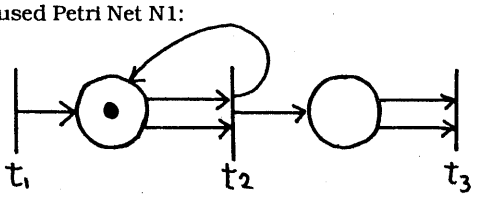
<Flat Synthesis>
It is assumed that the program has already been constructed from reusable software components up to this step. To start with, it is made clear what the inputs and outputs are :

INPUT:
Specification f
(written by LPTL)
INPUT:
Reused Programs N1
(represented by Petri net)
OUTPUT:
Tuned Programs N2
(represented by Petri net)

Synthesis Procedure (sketch)
This program synthesis method consists of the following three steps:
1) Build a linear structure G representing a model θ such that $\theta \in L(N1,f)$.
2) Transform G into an equivalent Petri net N_G.
3) Make a product Petri net N2 of N1 and N_G.

Example:

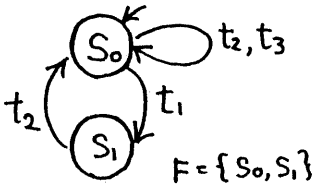
Reused Petri Net N1:



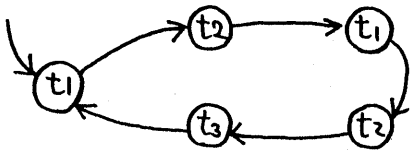
Specification f:

$$G(t1 \supset X t2) \wedge G(t1 \vee t2 \vee t3)$$

Buchi Sequential Automaton Af:



Linear Structure G:



Tuned Program N2:

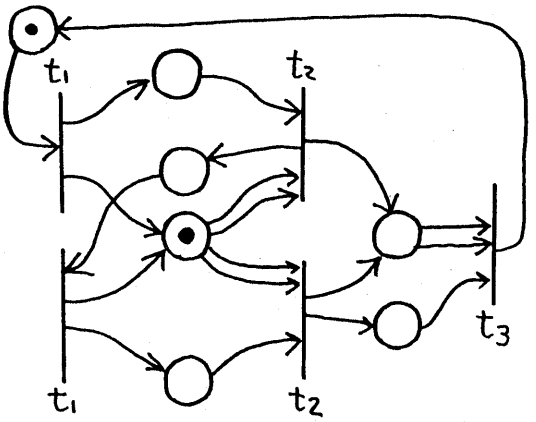


Fig.3 Flat Synthesis Example

The main drawback in this flat synthesis is that a synthesized program has no concurrency, because the program is serialized by a linear structure G.

To overcome this drawback, the compositional synthesis method is mentioned by introducing a module structure (Fig.4) into the Petri net, where an inside behavior of individual modules is serialized and becomes a sequential program. However, the modules can run concurrently with each other. This is summarized as illustrated in Fig.5.

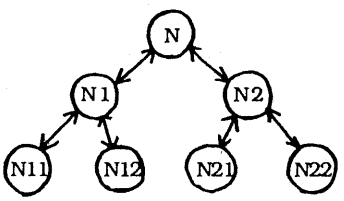
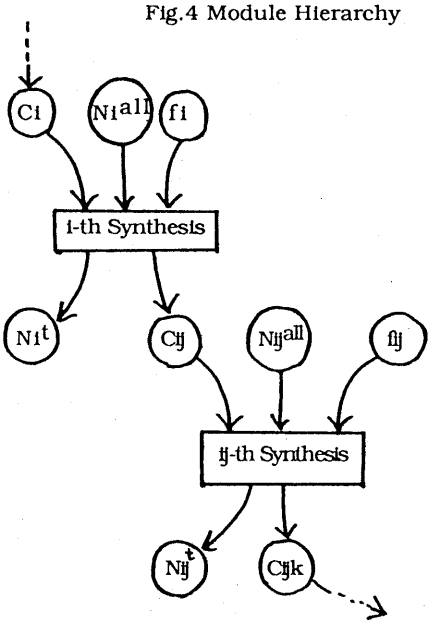


Fig.4 Module Hierarchy



Ni: Local Petri net for module i,
N_i^{all}: Global Petri net linking Ni,

N_i^t: Tuned Petri net after i-th synthesis

Ci: Constraint to i-th synthesis

fi: Temporal logic specification

Fig.5 Compositional Synthesis

<Compositional Synthesis>

i-th Synthesis Procedure (sketch)
1) Transfer Ci into an equivalent Petri net N_C, where Ci is a linear structure representing constraints to i-th synthesis that is generated by i-1-th synthesis procedure.
2) Make a product Petri net N2 of N_i^{all} and N_C.
3) Build a linear structure G representing a model θ such that $\theta \in L(N2,fi)$.
4) Reduce G into Cij focusing on module Nij.
5) Reduce G into GI focusing on module Ni.
6) Make a product Petri net N_i^t of N_i and G_I.

6. MENDEL/89

MENDEL/89 [UH89] is a concurrent programming language which is based on Petri net. MENDEL program consists of several objects that is a process. Behavior of an object is described as a set of methods, which intuitively looks IF-THEN rules. The skeleton of MENDEL/89 program is represented by MENDEL net, which is a Petri net with module structures (Fig.6). Therefore, it is possible to verify MENDEL programs using the proposed verification method. MENDEL/89 is powerful enough to describe practical concurrent programs such as a robot control system and the lift system. a MENDEL/89 program can be executed on the multi-personal computer system.

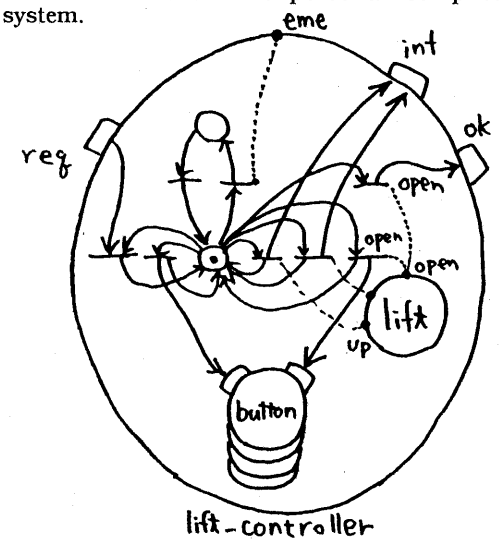


Fig.6 MENDEL Net Example

7. CONCLUSION

This research was carried out to establish a method to verify and synthesize concurrent programs automatically using the Petri net and temporal logic. In this paper, (1) we define the class combining Petri net and temporal logic which is decidable, (2) the decision procedure for this class is applied to concurrent program verification, and (3) a compositional synthesis method is provided by modifying reusable components to satisfy a specification. Our approach means relaxing automatic verification and synthesis for only finite-state

programs to infinite-state programs such as Petri net.

We have already implemented a programming synthesis system for finite-state programs (bounded Petri nets) which is called the MENDELS ZONE [UKMIH90]. This system consists of two major steps: (1) construction of a body part by reusable objects, and (2) synthesis of a synchronization part which is consistent with the body part, from a temporal logic specification and a bounded Petri net. We plan to extend this system so as to be based on this paper's method for infinite-state programs (unbounded Petri nets), after establishing reasonable efficiency for the synthesis algorithm.

At present, much still remains to be explored:
(1) In concurrent program verification, the branching time propositional temporal logic (BPTL) is more useful and practical than LPTL. It is important to develop a BPTL version.
(2) In the proposed synthesis method, a synthesized program may be unnecessarily deterministic, since it is synthesized from only one model satisfying the LPTL specification. To improve this demerit, it is necessary to introduce one of two program synthesis approaches. One is synthesis from a model generator in LPTL. The other is synthesis from a model in BPTL.

ACKNOWLEDGMENTS

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 and Takeshi Kohno of the Systems & Software Engineering Laboratory, TOSHIBA Corporation, for providing continuous support. We are also indebted to Fujio Umibe for proofreading and correcting the original English manuscript.

REFERENCES

[BP84] Biggerstaff,T.J. and Perlis,A.J., Special Issue on Software Reusability, Foreword, IEEE Trans. on SE, SE-10, No.5, 1984.
[CE82] 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.
[CES86] Clarke,E.M., Emerson,E.A., Sistla,A.P., Automatic Verification of Finite-state Concurrent Systems Using Temporal Logic Specifications, ACM TOPLAS, Vol.8,No.2,1986.
[CK87] Cherkasova,L.A. and Kotov,V.E., The Undecidability of Propositional Temporal Logic for Petri Nets, Computers and Artificial Intelligence 6, Vol.2, 1987.
[HR89] Howell,R. and Rolser,L.E. , On Questions of Fairness and Temporal Logic for Conflict-free Petri Nets, LNCS 340 Advances in Petri Nets 1988, 1989.
[KI82] Katai,o. Iwai,S., Construction of Scheduling Rules for Asynchronous, Concurrent Systems Based on Tense Logic (in Japanese), Trans. of SICE, vol.18,No.12,1982.
[KG87] Kaiser,G. and Garlan,D., Melding Software Systems from Reusable Building Blocks, IEEE software, Vol.4, No.2, 1987.
[Mu89] Murata,T., Petri Nets: Properties, Analisys and Applications, Proceedings of the IEEE, Vol.77,No.4,1989.
[MW84] 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.
[Pe81] Peterson,J.L., Petri Net Theory and the Modeling of Systems, Prentice-Hall, Inc.,1981.
[Pn77] Pnueli,A., The Temporal Logic of Programs, Proc. of 18th FOCS, 1977.
[SCFM84] Sistla,A.P., Clarke,E.M., Frances,N., Meyer,A.R., Can Message Buffers Be Axiomatized in Linear Temporal Logic?, Information and Control 63, 1984.
[SL89] Suzuki,I., and Lu,H., Temporal Petri Nets and Their Application to Modeling and Analisys of a Handshake Daisy Chain Arbiter, IEEE Trans. on Computer, Vol.38,No.5,1989.
[UH89] Uchihira,N., Honiden,S. , Petri Net-based Concurrent Programming Language on Distributed Processing Systems (in Japanese), EIC CPSY89-34, 1989.
[UKH89] Uchihira,N., Kawata,H, Honiden,S. , A Concurrent Program Synthesis Using Petri Net and Temporal Logic in MENDELS ZONE, ICOT TR-449, 1989.
[UKMH87] Uchihira, N., et al., Concurrent Program Synthesis with Reusable Components Using Temporal Logic, Proc of COMPSAC87, 1987.
[UKS89] Uchihira,N., Kawata,H, Sumida,S. , Design Support System for Hierarchical Distributed Systems (in Japanese), 30th Programming Symposium, 1989.
[VJ85] Valk,R. and Jantzen,M., The Residue of Vector Sets with Applications to Decidability Problems in Petri Nets, Acta Informatica 21, 1985.
[VW86] Vardi,M. Wolper,P. ,An Automata-Theoretic Approach to Automatic Program Verification, Proc. 1st Symp. on Logic in Computer Science, 1986.
[Wo89] Wolper,P., On the Relations of Programs and Computations to Models of Temporal Logic, in Proc. of the 1987 Manchester Workshop on Temporal Logic, Springer-Verlag LNCS vol.398, 1989.
[WVS83] Wolper,P., Vardi,M.Y.,Sistla,A.P., Reasoning about infinite Computation Paths, Proc. of 24th FOCS, 1983.