ICOT Technical Memorandum: TM-0900

TM-0900

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

by
N. Uchihira & S. Honiden (Toshiba)

July, 1990

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

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

Naoshi Uchihira Shinichi Honiden

Systems & Software Engineering Laboratory
TOSHIBA Corporation
Yanagi-cho 70, Saiwai-ku, Kawasaki 210 JAPAN
Phone: (044)548-5465 E-mail: uchi@ssel.toshiba.co.jp

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 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 formulated as an infinite language and whose satisfiability problem is decidable. We then show how to verify concurrent programs using Petri nets and temporal logic, and also propose a compositional synthesis method that can tune up a row program (reused program) to satisfy a temporal logic specification.

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 the 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 can 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 satisfiability (emptiness) problem. In this paper, we select a class that is decidable in section 3. In that class, a transition firing sequence corresponds to a model of temporal logic formula, and we can combine Petri net and temporal logic as infinite Petri net languages. Infinite Petri net languages are well investigated by Valk [Va83,VJ85]. The following results will be shown using techniques of infinite Petri net language: (1) It is decidable whether a Petri net satisfies a propositional temporal logic specification, and (2) for given Petri net N and propositional temporal logic specification f, the new Petri net N' can be constructed by modifying N such that N' satisfies f. We apply these results to verification and compositional synthesis for concurrent programs, in Sections 4 and 5, respectively. Our verification method allows us to verify properties such as mutual exclusion and partial ordering of events. It promises to complement the traditional Petri net analysis methods. Our synthesis method can tune up an original concurrent program such as a robot control program to satisfy a given temporal specification in the compositional way.

2. PRELIMINARIES

Definition 1 (Petri net) [Mu89]:
A Petri net is a 5-tuple, $N=(P,T,F,w,m_0)$ where:
$P = \{p_1, p_2, ..., p_n\}$ is a finite set of places,
$T = \{t_1, t_2, ...t_m\}$ is a finite set of transitions,
$F \subset (P \times T) \cup (T \times P)$ is a set of arcs (flow relations),
$w: F \rightarrow \{1,2,3,...\}$ 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.

Let t be a transition and P={p1,p2,..,pn}. We define a n-dimensional differential vector $\Delta(t)=(w(t,p1)-w(p1,t),..., w(t,pn)-w(pn,t))$. Furthermore, $\Delta(t1t2...tm) = \Delta(t1) + \Delta(t2) +...+ \Delta(tm)$ for a transition sequence t1t2...tm.

Definition 2 (Product of Petri nets)
For given Petri nets $N1=(P1,T1,F1,w1,m_{01})$ and $N2=(P2,T2,F2,w2,m_{02})$ where $P1 \cap P2=\emptyset$, $N=N1 \times N2$ is a product of N1 and N2 such that $N=(P,T,F,w,m_0)$, $P=P1 \cup P2$, $T=T1 \cup T2$, $F=F1 \cup F2$, $w(p,t)= w1(p,t) + w2(p,t)$, $w(t,p)= w1(t,p) + w2(t,p)$, $m_0=m_{01} \cup m_{02}$.

Example:

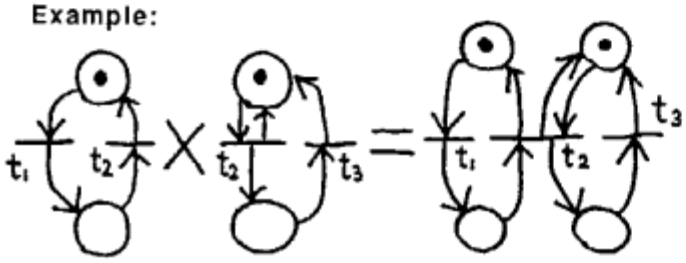


Fig. 1 Product of Petri Net

Definition 3 (Labelled Petri net)
A labelled Petri net is a 6-tuple, N = (P, T, F, w, h, m_0) 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 $X \subset \Sigma$ be a finite set, 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, $X^\infty \equiv X^* \cup X^\omega$. 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 (P,T,F,w,m_0), if the legal 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 4 (Petri net language)
L(N) is a Petri net language generated from Petri net N if $L(N) = \{h(\theta) \mid \theta \in F(N)\}$, and $L_\omega(N)$ is a Petri net ω language generated from Petri net N, if $L_\omega(N) = \{h(\theta) \mid \theta \in F_\omega(N)\}$

When $\theta \in T^*$ and $T' \subset T$, we define $\theta/T' \equiv h(\theta)$ where h(t)=t if $t \in T'$, otherwise $h(t)=\lambda$. Also, $L(N)/T' \equiv \{\theta/T' \mid \theta \in L(N)\}$. This means $t \in T'$ is visible and $t \in T-T'$ is invisible.

Definition 5 (Fair Petri net language)
$L_\omega^{fair}(N) \subset L_\omega(N)$ is a fair infinite Petri net language from Petri net N which supposes the fairness condition that whenever a transition t is infinitely enabled then t will eventually fire.

Definition 6 (Linear time propositional temporal logic):
(1) Syntax
Linear time propositional temporal logic (LPTL) formulas are built from
- A set of all atomic propositions: Prop={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 7 (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 $s_i \in \rho(s_{i-1}, t1)$ 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 [WVS83]:

Given an LPTL formula f, one can build a Büchi sequential automation $A_f=(\Sigma,S,\rho,s0,F)$, where $\Sigma=2^{Prop}$, such that $L(A_f)$ is exactly the set of sequences satisfying formula f.
Proof. Omitted. ■

Definition 8 (Single event condition) [MW84]:
A single event condition is
$G((\bigvee_{1\le i\le n} p_i) \wedge (\bigwedge_{1\le i<j\le n} \neg(p_i \wedge p_j))$,
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 $A_{f'} = (\Sigma,S,\rho,s0,F)$ where f' is f with a single event condition, we can make $\Sigma=Prop$ in place of $\Sigma=2^{Prop}$, because only one atomic proposition is true at each time.

Example:
The following $A_{f'}$ is built from LPTL formula $f = G(t1 \supset X(\neg t1\ U\ t2) \wedge G(t2 \supset X(\neg t2\ U\ t1)$ with a single event condition:
$A_{f'}= (\{t1,t2\},\{s0,s1,s2\},\rho,s0,\{s0,s1,s2\})$ where
$\rho = \{\{s1\}=\rho(s0,t1),\ \{s2\}=\rho(s0,t2),\ \{s0\}=\rho(s1,t2),\ \{s0\}=\rho(s2,t1)\}$

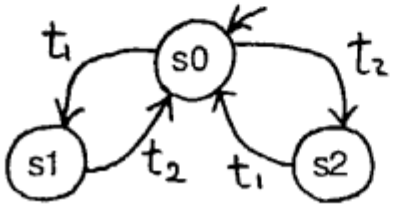


Fig. 2 Temporal Logic and Automaton

Definition 9:
Ls(f) is a ω language generated from an LPTL formula f with a single event condition if $Ls(f) = L(A_{f'})$ where f' is f with a single event condition and Prop becomes an alphabet of $A_{f'}$.

Lemma 1:
Given an LPTL formula f, $Ls(f)^C = Ls(\neg f)$, where $Ls(f)^C = Prop^\omega - Ls(f)$.
Proof.
In [WVS83], it is proved that $L(A_{f1})^C = L(A_{f2})$, where f2 $= \neg f1$ 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 propositions 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.
Note fi(t) $\supset$ en(t) always holds.

Emptiness problem is roughly defined as to decide whether for a given Petri net N and a given temporal logic specification f, there exists a legal firing transition sequence on N satisfying f.

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
[UKMIH90]	d	bounded	decidable

Table 1 Several Combinations of Petri Net and Temporal Logic

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 $L(A_f)$ where A_f 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 $h: T\to\Sigma$ is defined such that $h(t)=t$, if $t \in T$ is visible, and $h(t)=\lambda$, if $t \in T$ is invisible. We are now going to define a new formal language from L(N) and Ls(f).

Definition 10:

Let $N = (P,T,w,m_0)$ and $Prop \subset T$ be a set of visible transitions which appear in a temporal logic formula f. We define $L(N,f) = L\omega(N) \cap Ls(f)$ where $L\omega(N)$ is a language generated from a labeled Petri net (P,T,w,h,m_0) with $h: T\to Prop$, and Ls(f) is a language generated from f under a single event condition.

Theorem 2:
For a given Petri net N=(P,T,w,m0) and LPTL formula f composed of a set of atomic propositions Prop, the emptiness problem of L(N,f) (i.e. L(N,f) is empty or not) is decidable.

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

Main Procedure

1) A Büchi sequential automation $A_f=(Prop,S,\rho,s_0,F)$ accepting Ls(f) is constructed according to [Wo89]. Here, Prop $\subset$ T.
2) Then construct a extended coverability graph G from N and A_f. G is a labeled directed graph. Each node x of G is represented as a k+2-tuple x=(x1,...,xk,s,f) where xi $\in \{0,1,...\} \cup \{\omega\}$ $(1 \le i \le k)$, s $\in$ S, f $\in$ {0 (normal node), 1 (designated node)}. Each edge e=(x,y) is labelled with an element of T. We define t $\in$ T is *enabled* in x if t is enabled at a marking (x1,...,xk) and $\rho(s,t) \neq \emptyset$, and t is *local enabled* if t is enabled in the marking and t $\notin$ Prop. G is constructed as follows:
2-1) Start with a graph G containing only an initial node $q0=(x^0_1,...,x^0_k,s0,f)$ where $m0=(x^0_1,...,x^0_k)$, s0 is an initial state of A_f and f=1 if s0 $\in$ F otherwise f=0. Let q0 be an unapplied node.
2-2) Repeatedly apply the following Graph Addition Procedure to the new (unapplied) nodes of G until all nodes of G have been applied.

Graph Addition Procedure
1) Let x be a given node with x=(x1,..,xk,s,f). Create new nodes x' = (x'1,...,x'k,s',f') from x according to (a)-(d) for all enabled transitions t at x and all s' $\in \rho(s,t)$, and also create new nodes x'=(x'1,...,x'k,s,0) according to (a)-(c) for all local enabled transitions t:
(a) x'i= ω if xi= ω $(1 \le i \le k)$.
(b) If there is a node y = (y,...,yk, s', fy) on some path from q0 to x (that ts an ancestor of x) such that $yj \le xj-w(pj,t)+w(t,pj)$ for all j $(1 \le j \le k)$ and $yi<xi-w(pi,t)+w(t,pi)$ for some i, then x'i=ω.
(c) For other i, x'i=xi-w(pi,t)+w(t,pi).
(d) f=1 if s'$\in$ F, otherwise f=0.
2) If x' is new in G, that is G doesn't have the same node, add a new node x' and a new edge e=(x,x') labelled with t, otherwise add only a new edge e=(x,x') labelled with t.

Above procedure always terminates, because G is finite as same as the normal coverability graph of a Petri net.
Claim: $L\omega(N) \cap Ls(f) \neq \emptyset$ iff there exists a cycle $c=x_0x_1...x_kx_0$ on CG such that x_0 is a designated node (i.e. f=1) and $\Delta(\theta) \ge 0$ where $\theta=t_1...t_{k+1}$ is a transition sequence over c (i.e. ti=e(xi-1,xi)). Note that θ is not necessary to be *legal*. The above claim follows directly from the result (Lemma5.4) of [SCFM84]. Furthermore, it is decidable if there exists such a cycle as follows [VJ85]: For each designated node x0, a set of all transition sequences θ which forms any cycle on G passing through x0 can be represented by a regular expression R. We want to decide if there are some $\theta \in$ R such that $\Delta(\theta) \ge 0$. For this purpose, we can regard R as commutative. Therefore, R can be expressed as finite sum of terms of the shape $\theta1^*\theta2^*...\theta n^*$ using the decomposition rules ($AB=BA$, $A^*B^*=(AB)^*(A^*+B^*)$, $(A+B)^* = A^*B^*$, $(A^*B)^*=1+A^*B^*B$) [Co71]. For each $\theta1^*\ \theta2^*...\theta n^*$, we can effectively decide whether $\Delta(\theta1^*\theta2^*...\theta n^*) = \alpha1\Delta(\theta1) + \alpha2\Delta(\theta2) + ... \ \alpha n\Delta(\theta n) \ge 0$ for some $\alpha1, \alpha2, ..., \alpha n \ge 0$. ■

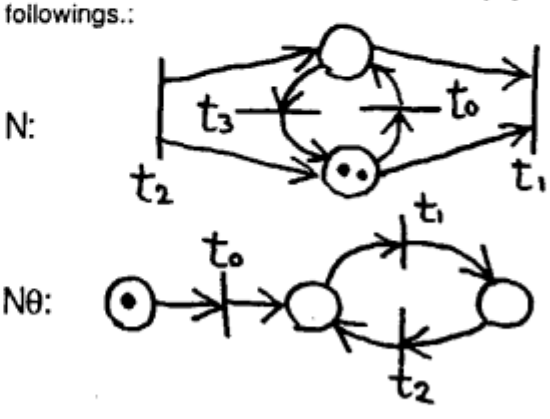
When L(N,f) is nonempty, it is very important to find a concrete sequence accepted by L(N,f) for the sake of program synthesis.

Theorem 3:
Given L(N,f) that is nonempty, we can construct a deterministic transition sequence $\theta_0\theta_c^{\omega} \in L(N,f)$.
Proof. omitted. ■

Furthermore, we can construct a new Petri net which reproduces transition firing $\theta_0\theta_c^{\omega}$ by tuning the original net (i.e. adding several places, transitions, arcs to the original net).

Theorem 4:
For given $N=(P,T,F,w,m_0)$ and LPTL formula f with a set of atomic propositions Prop, if L(N,f) that is nonempty, we can construct a Petri net $N'=(P',T',F',w',m_0')$ such that N' is dead-lock free, $L\omega^{fair}(N')/Prop = \{\theta_0\theta_c^{\omega}\}$, P'$\supset$P, T'$\supset$T and F'$\supset$F.
Proof. Firstly we can easily construct a Petri net Nθ such that $L(N\theta)=\{\theta_0\theta_c^*\}$. Then, make a product Petri net Np = N $\times$ Nθ. Finally, we can construct N' such that N' is deadlock-free and there exists a legal firing sequence from all reachable markings to any t appearing in θ_c, by tuning up Np according to the Valk and Jantzen's tuning method [VJ85] (cf. appendix). It is clear $L\omega^{fair}(N')/Prop = \{\theta_0\theta_c^{\omega}\}$ because N' can reproduces $\theta_0\theta_c^{\omega}$ under the fairness condition. ■

Example:
Let θ_0=t0 and θ_c=t1t2 and $L(N\theta)=\{\theta_0\theta_c^*\}$. N', Np and N$\theta$ are constructed from given N and $\theta_0\theta_c^{\omega}$ as the followings.:



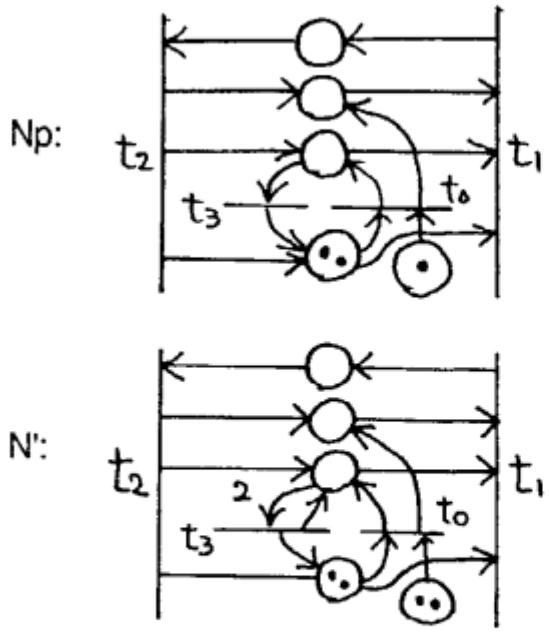


Fig. 3 Tuning Method

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_ω (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. 4.

Fig.4 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 11:
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 5:
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 2. ■

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 specifications, 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.

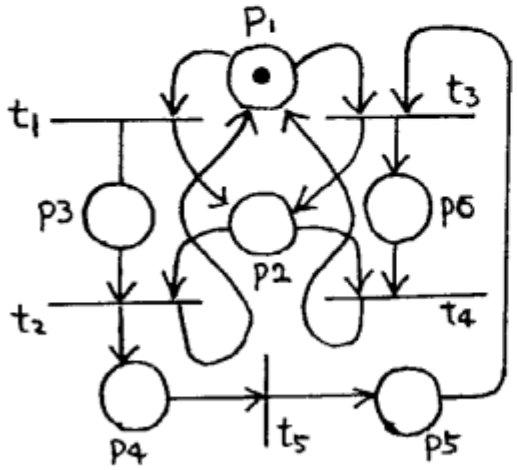
However, the blind side of this verification can be complemented by the traditional analysis method (liveness problem, boundedness problem, etc) for Petri nets. The combination of both verification methods is effective.

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. 5, where places P4 and P5 are unbounded buffers. And specification f is given

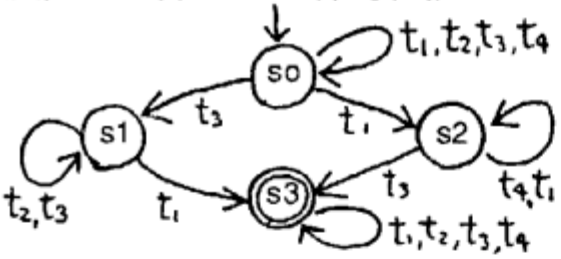
that intervals [t1,t2] and [t3,t4] satisfy a mutual exclusion condition as follows:

Specification f:
$f = G(t1 \supset X(\neg t3\ U\ t2)) \wedge G(t3 \supset X(\neg t1\ U\ t4))$

Petri net N :
(This Petri net is deadlock free itself, therefore we ignore **nop** for simple explanation.)



Büchi sequential automaton A¬f:
$A_{\neg f}=(\{t1,t2,t3,t4\},\{s0,s1,s2,s3\},\rho,s_0,\{s3\})$



Extended coverability graph G:

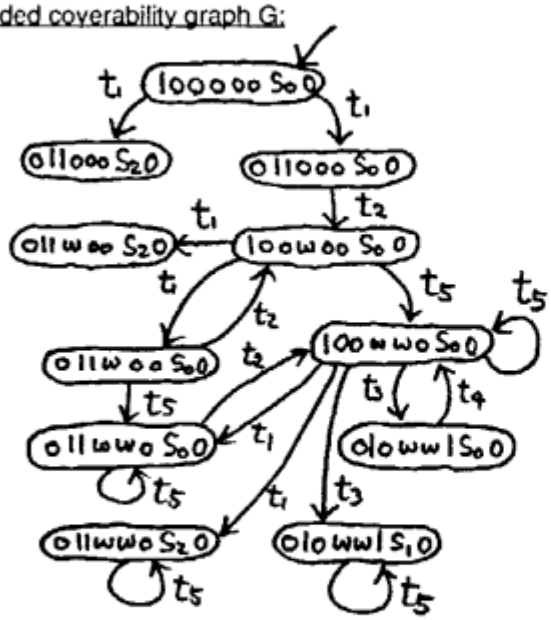


Fig. 5 Example of Verification

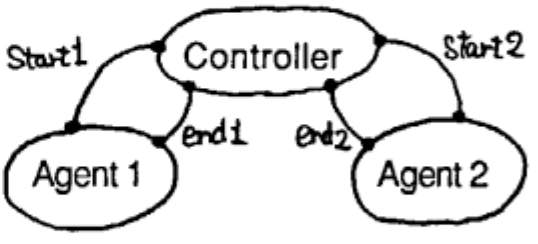
In G, there exists no designated node, that means $L_S(\neg f) \cap L\omega(N\omega) = \emptyset$ from Theorem 2. We conclude Nω satisfies f.

5. COMPOSITIONAL PROGRAM SYNTHESIS

It is not easy for a ordinary programmer to realize a correct synchronization in concurrent programs, and it requires tremendous debugging efforts. This section provides a method to synthesize automatically a concurrent program with reusable components by program tuning. The goal programs are synthesized to satisfy the given specification by tuning up reused programs that are represented by Petri nets. This method differs from other synthesis methods [MW84,CE82] that also use the temporal logic specification, in the point of utilizing software reuse. We also emphasize our method adopts a compositional way to synthesize. It is necessary for two reasons: (1) Reusable software itself is composed compositionally in a ordinary software, and (2) global synthesis of a large-scale program requires huge then unpractical computing power. The model building techniques in Theorem 3 and Theorem 4 are used in this synthesis method.

5.1 Concurrent Program Structure

It is assumed that a target program consists of one controller and several agents. While the controller controls each agent sequentially, the agent is independent with other agents and can run concurrently each other. This structure is very natural in some domain such as robot control systems and plant control systems. An example is shown in Fig.6. The controller and the agent i communicate with each other by a set of synchronous communication channels Ti, like CCS [Mi89]. It is assumed that a raw controller and raw agents has already been constructed from reusable software components up to this step. Here, the raw controller is represented by Petri net $Nc=(Pc,Tc,Fc,wc,mc_0)$, and the raw agent i is represent by $Na_j=(Pa_j,Ta_j,Fa_j,wa_j,ma_{j0})$. Note that a set of channels $Ti = Tc \cap Ta_j$. In case of this example (Fig.6), a controller and an agent may be represented as shown in Fig.7.



T1={start1,end1} T2={start2,end2}

Fig.6 Concurrent Program Structure

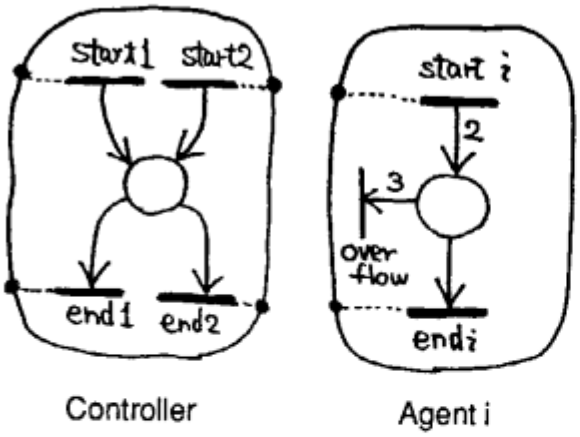


Fig.7 Raw Controller and Raw Agent represented by Petri net

5.2 Temporal Logic Specification

User specifies several constraints by a LPTL formula f with a set of atomic propositions Prop=Tc so that the controller satisfies f cooperating with all agents.

Example:

Prop=Tc={start1,end1,start2,end2}

$f=G(start1 \supset X(\neg start2\ U\ end1)$
$\wedge\ G(start2 \supset X(\neg start1\ U\ end2)$

The above f means that once agent 1 starts, Agent 2 never starts until Agent 1 ends, and also Agent 2 starts, Agent 1 never starts until Agent 2 ends.

Here, a concurrent program synthesis means to tune up reusable components to satisfy these specifications (constraints). To start with, it is made clear what the inputs and outputs are:

INPUT:
Specification f
(written by LPTL)

INPUT:
Reused Programs
One raw Controller and several raw Agents
(represented by Petri net Nc, $Na_1, Na_2, ..., Na_k$)

OUTPUT:
Synthesized Programs
One Controller and several Agents
(represented by Petri net Nc', $Na_1', Na_2', ..., Na_k'$)

First, we show the controller synthesis method and then the agent synthesis method. This compositional way is highly practical to synthesize large-scale programs.

5.3 Controller Synthesis

This controller synthesis method consists of the following four steps:

1) Each Petri net Na_i of agent i is reduced as possible [LF85] into Na_i^r with $L(Na_i)/T_i = L(Na_i^r)/T_i$.

2) Make a product Petri net $N = Nc \times Na_1^r \times ... \times Na_k^r$.

3) Construct a infinite firing sequence $\theta=\theta_0\theta_c^\omega \in L(N,f)$. from Theorem 3.

4) Construct a Petri net Nc' such that $L(Nc')=\{\theta_0\theta_c^*\}$.

Nc' is a Petri net of synthesized controller, that is a deterministic sequential program.

Example: Nc' is synthesized from a transition sequence θ = (start1 end1 start2 end2)$^\omega$.

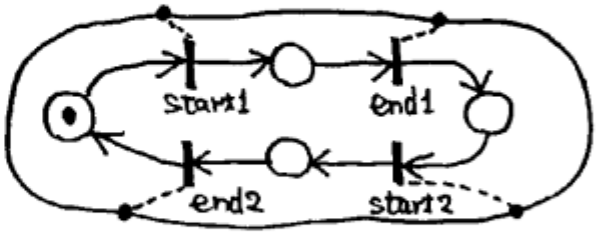


Fig. 8 Synthesized Controller

5.4 Agent Synthesis

For each agent, we can construct a tuned agent Petri net Na_i' from Na_i and $\theta_0\theta_c^\omega$ as follows:

1) Construct a Petri net $N\theta_i$ such that $L(N\theta_i)=\{\theta_0\theta_c^*/T_i\}$.

2) Make a product Petri net $Na_i^p = Na_i^r \times N\theta_i$.

3) Tune up Na_i^p into Na_i' using the Valk and Jantzen's method. (cf. Theorem 4)

Example:

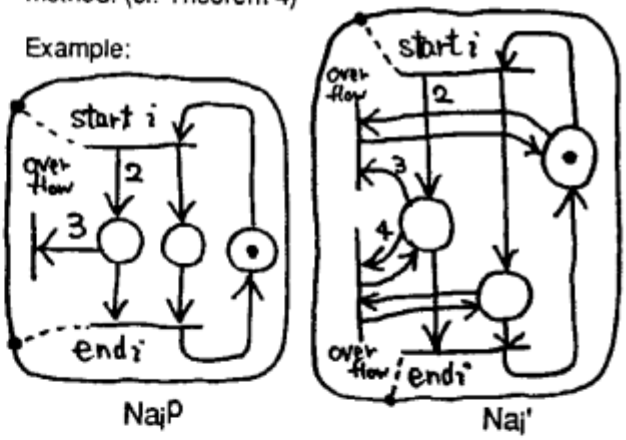


Fig. 9 Synthesized Agent

Note that the synthesized controller is deterministic sequential program while the tuned agents can be nondeterministic concurrent programs. Here we must assume a fairness condition for each agent that if a transition t is infinitely enabled then t will sometime fire.

Theorem 6:
If Petri net Nc', $Na_1', Na_2', ..., Na_k'$ are synthesized from Petri net Nc, $Na_1, Na_2, ..., Na_k$ and LPTL formula f according to the above synthesis method, then $N'=Nc' \times Na_1' \times Na_2' \times ... \times Na_k'$ is deadlock free and

$L\omega^{fair}(N')/Prop = \{\theta_0\theta_c^\omega\} \subset L(N,f)$.

Proof. Omitted. ■

The main drawback in this synthesis is that a synthesized controller is deterministic, because the program is serialized by a deterministic firing sequence. However,when expanding a deterministic one to nondeterministic one that may be more natural, it is indispensable to consider invisible transition of each agent, which requires other information besides given communication channels. That is estimated to decrease concurrency of a synthesized program.

6. 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. The significant feature of our approach is to relax automatic verification and synthesis for only finite-state programs [MW84,CES86] to infinite-state programs such as Petri net.

This research has been supported by ICOT.

REFERENCES

[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.

[CES83] 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.

[Co71] Conway,J.H., Regular Algebra and Finite Machines, Chapman and Hall, 1971.

[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.

[LF85] Lee,K.H., Favrel,J., Hierarchical Reduction Method for Analysis and Decomposition of Petri Nets, IEEE Trans. on SMC, Vol.SMC-15, No.2, 1985.

[Mi89] Milner,R., Communication and Concurrency, Prentice Hall, 1989.

[Mu89] Murata, T., Petri Nets: Properties, Analysis 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 Analysis of a Handshake Daisy Chain Arbiter, IEEE Trans. on Computer, Vol. 38, No. 5, 1989.

[UKMIH89] Uchihira, N., et al., Concurrent Program Synthesis: Automated Reasoning Complements Software Reuse, Proc. of IEEE 23rd Hawaii International Conference on System Science, 1990.

[Va83] Valk, R., Infinite behavior of Petri nets, Theoret. Comput. Sci. 25, 1983.

[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.

Appendix

Valk and Jantzen's method [VJ85]

Definitions.

Σ N=(P,T,F,w,m_0) is a Petri net.

Σ A marking m is T'-continual, if an infinite sequence of transitions can fire (i.e. be regal) in m containing each t $\in$ T' $\subset$ T infinitely often.

- CONTINUAL(T') $\int$ {m| m is T'-continual}.
- $\mathbf{N}$ is a set of non negative integers. Let K $\subset$ $\mathbf{N}^k$, then the residue set of K, written res(K), is the smallest subset of K which satisfies res(K)+$\mathbf{N}^k$=K+$\mathbf{N}^k$.

Theorems.

- res(CONTINUAL(T')) is finite and can be effectively constructed.
- Using res(CONTINUAL(T')), we can construct a new Petri net N' whose all reachable markings are lying in CONTINUAL(T') with the same number of places of N, but possibly additional transitions.