A Formal Method for Analyzing Software Architecture Models in SAM *

Huiqun Yu †, Xudong He, Yi Deng, Lian Mo
School of Computer Science
Florida International University
Miami, FL 33199, USA
{yhq|hex|deng|lmo01}@cs.fiu.edu

Abstract

The Software Architecture Model (SAM) is a general software architecture model based on a dual formalism combining Petri nets and temporal logic. A SAM model contains a hierarchical set of compositions, each of which consists of a set of components, a set of connectors, and a set of constraints. This paper proposes a formal method for analyzing SAM models in both element (either component or connector) level and composition level. The basic idea is to simulate Petri net behaviors in terms of fair transition systems. The properties of individual components and connectors are verified either by deductive reasoning or model checking. The properties of the entire system is inferred from the properties of its constituents. A detailed case study of an electronic commerce system shows our approach to formally modeling, refining and analyzing software architecture models.

Keywords: *Software architecture, SAM, Petri net, temporal logic, verification*

1 Introduction

SAM is a general software architecture model for developing and analyzing software architecture specifications [14], and has been developed in the past 4 years at the Florida International University. The theoretical basis of SAM is a combination of two complementary formal notions: predicate transition nets (PrTNs for short) [4] and linear time first order temporal logic [9]. PrTNs are used to define the behavior models of components and connectors, and temporal logic is used to specify properties (or constraints) of components and connectors. The sound mathematical foundation of SAM can help to detect and eliminate design errors early in the development cycle, avoid costly fixes at the implementation stage, and thus to reduce overall development cost and to increase the quality of the system.

Formal verification is a promising way to ensure the correctness of software products [3]. However, the task is not trivial except for toy programs [1]. To ensure the correctness of large systems, it's extremely challenging. It has long been understood that the only hope for verifying a large program is by decomposing it into many small modules, and establishing properties of each individual module while considering the rest of the processes as its environment.

This paper proposes such a way to formally analyze software architecture models in SAM. The basic idea for element verification is to simulate Petri net behaviors in terms of fair transition systems. Properties of the systems are verified either by deductive reasoning or model checking. The verification can be done interactively using the Stanford Temporal Prover tools (STeP) [8]. For composition level, we infer properties of the entire program from the properties of its constituents. The method accords well with the SAM methodology for software architecture modeling and development.

The rest of the paper are organized as follows: Section 2 gives a brief introduction to SAM and its theoretical basis. Section 3 proposes our verification method. Section 4 presents a detailed case study which shows how to apply our method to formally modeling, refining and analyzing software architecture models. Section 5 is the conclusion.

2 The basis of SAM

In SAM, a software architecture is defined by a hierarchical set of compositions, in which each composition consists of a set of components, a set of connectors, and a set

*Supported in part by the NSF under grants HDR-9707076 and CCR-0098120, and by NASA under grant NAG 2-1440.

†On leave from East China University of Science and Technology, Shanghai.

of constrains to be satisfied by the interacting components. All of components and connectors are described by a dual formalism of PrTNs and temporal logic. In this paper, we use the conventions in [6] for PrTNs, and the conventions in [9] for temporal logic.

2.1 The PrTNs

A *PrTN* consists of (1) a finite net structure (P, T, F), (2) an algebraic specification *SPEC*, and (3) a net inscription (φ, L, R, m_0). (P, T, F) is the essential net structure, where $P \cup T$ is the set of nodes satisfying the condition $P \cap T = \emptyset$. P is called the set of *places* (graphically represented by circles) and T is called the set of *transitions* (represented by boxes). F is the set of arcs and is called the *flow relation* satisfying the condition: $F \subseteq P \times T \cup T \times P$.

The algebraic specification *SPEC* is a meta-language to define the tokens, labels, and constraints of a PrTN. The underlying specification *SPEC* = (S, *OP*, *Eq*) consists of a signature **S** = (S, *OP*) and a set *Eq* of **S**-equations. Signature **S** = (S, *OP*) includes a set of sorts S and a family *OP*= $(OP_{s1,...,sn,s})$ of sorted operations for s_1, ..., s_n, $s \in S$. For each $s \in S$, we use CON_s to denote $OP_{,s}$ (the 0-ary operation of sort s), i.e. the set of constant symbols of sort s. The **S**-equations in *Eq* define the meanings and properties of operations in *OP*. We often simply use familiar operations and their properties without explicitly listing the relevant equations.

Tokens of a PrTN are ground terms of the signature **S**, written MCON_S. The set of labels is denoted by $Label_S(X)$ (X is the set of sorted variables disjoint with *OP*). Each label can be a multiple set expression of the form $\{k_1x_1$, ..., $k_nx_n\}$. Constraints of a PrTN are a subset of first order logic formulas (where the domains of quantifiers are finite and any free variable in a constraint appears in the label of some connecting arc of the transition), and thus are essentially propositional logic formulas. The subset of first order logical formulas contains the S-terms of sort *bool* over X, denoted as $Term_{OP,bool}(X)$.

The net inscription (φ, L, R, m_0) associates each graphical symbol of the net structure (P, T, F) with an entity in *SPEC*, and thus defines the static semantics of a PrTN. Each place in a PrTN is a data structure and a component of the overall system state. The sort of each place (a member of S in *SPEC*) defines its valid values, i.e. proper tokens. Therefore, we associate each place p in P with a subset of sorts in S, and give the sort assignment $\varphi : P \to \wp(S)$. Mapping L: $F \to Label_S(X)$ is a sort-respecting labeling of the PrTN. Mapping R: $T \to Term_{OP,bool}(X)$ is a well-defined constraining mapping of N, which associates each transition t in T with a first order logic formula defined in the underlying algebraic specification.

A *marking* m of a PrTN is a mapping, $P \to MCON_S$, from the set of places to multi-sets of tokens. An *occurrence mode* of a PrTN is a substitution $\alpha = \{x_1 \leftarrow c_1, \ldots, x_n \leftarrow c_n\}$, which instantiates typed label variables. We use $e : \alpha$ to denote the result of instantiating an expression e with α, in which e can be either a label expression or a constraint. Given a marking m, a transition $t \in T$, and an occurrence mode α, t is *enabled* if the following condition is satisfied : $\forall p : p \in P.(L(p,t) : \alpha \subseteq m(p)) \wedge R(t) : \alpha$

The *firing* of an enabled transition t at marking m and occurrence mode α returns the marking m' defined by $m'(p) = m(p) - L(p,t) + L(t,p)$, for all $p \in P$. We call m' the t-successor of m, denoted by mtm'. A *run* σ for PrTN π in initial mark m_0 is an infinite sequence of markings: $\sigma = m_0, m_1, m_2, \ldots$, for each pair of markings (m_i, m_{i+1}), there is an enabled transition t_i such that $m_i t_i m_{i+1}$.

Given a PrTN π and an initial marking m_0, we define the *computation* of π with the initial marking m_0 to be the set of all possible runs. We use $Comp(\pi)$ to denote the computation of a PrTN π.

An example (the dining philosophers) Figure 1 shows a PrTN which models the well-known dining philosophers' problem.

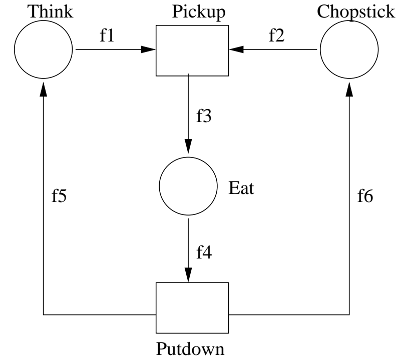

Figure 1. A PrTN model of the dining philosophers

There are three places (*Think*, *Chopstick* and *Eat*) and two transitions (*Pickup* and *Putdown*) in the PrTN. In the underlying specification *SPEC* = (S, *OP*, *Eq*), where S includes elementary sorts such as Integer and Boolean, and also sorts PHIL and CHOP derived from Integer. S also includes structured sorts such as set and tuple obtained from the Cartesian product of the elementary sorts; *OP* includes standard arithmetic and relational operations on Integer, logical connectives on Boolean, set operations, and selection operation on tuples; and *Eq* includes known properties of the above operators.

The net inscription (φ, L, R, m_0) is as follows:

- Sorts of places:

  φ (Think) = $\wp$(PHIL),

φ(Eat) = $\wp$(PHIL×CHOP×CHOP),
φ(Chopstick) = $\wp$(CHOP)

- Arc definitions:
  L(f1) = {ph}, L(f2) = {<ch1,ch2>},
  L(f3) = {<ph,ch1,ch2>}, L(f4) = {<ph,ch1,ch2>},
  L(f5) = {ph}, L(f6) = {<ch1,ch2>}
- Constraints of transitions:
  R(Pickup) = (ph = ch1) $\wedge$ (ch2 = ph $\oplus$ 1),
  R(Putdown) = true
- The initial marking m_0 is defined as follows:
  m_0(Think) = {1, 2, ..., K}, m_0(Eat) = { }, m_0(Chopstick) = {1, 2, ..., K}.

2.2 Temporal logic

The requirement specification language is linear temporal logic (LTL). A *temporal formula* is constructed out of state formulas to which we apply the boolean connectives and temporal operators. Future temporal operators include $\Box$ (always), $\mathcal{W}$ (wait-for), $\Diamond$ (eventually), $\bigcirc$ (next), and $\mathcal{U}$ (until). Past operators include $\ominus$ (previously), $\boxminus$ (so-far), $\diamondminus$ (once), and $\mathcal{S}$ (since). We write $p \Rightarrow q$ as an abbreviation of $\Box(p \rightarrow q)$. A *past formula* is one that contains no future temporal operators.

A *model* for a temporal formula p is an infinite sequence of states $\sigma : s_0, s_1, \ldots$, where each state s_j provides an interpretation for the variables mentioned in p. The semantics of a temporal formula p in a given model σ and position j is denoted by $(\sigma, j) \models p$. For example,

- For a state formula p, $(\sigma, j) \models p \quad \Leftrightarrow \quad s_j \models p$
- $(\sigma, j) \models \Box p \quad \Leftrightarrow \quad (\sigma, i) \models p$ for all $i \geq j$
- $(\sigma, j) \models \Diamond p \quad \Leftrightarrow \quad (\sigma, i) \models p$ for some $i \geq j$

A model $\sigma : s_0, s_1, \ldots$ satisfies a temporal formula if $(\sigma, 0) \models p$. A temporal formula p that is valid over a program P specifies a property of P, i.e., states a condition that is satisfied by all computations of P.

Since LTL is an extension of first-order logic, the deductive system for LTL includes axioms and rules from first-order logic, as well as those axioms and rules specific to temporal operators. Typical rules include:

- Modus ponens: $p, p \rightarrow q \vdash q$
- Chain: $p \rightarrow \Diamond q, q \rightarrow \Diamond r \vdash p \rightarrow \Diamond r$

The interested reader is referred to [9, 10] for detailed presentation of temporal logic and verification techniques.

Two important classes of properties are: *safety* and *response*.

- Safety properties are those that can be expressed by a formula $\Box p$, for some past formula ψ.
- Response properties are those that can be expressed by a formula $p \Rightarrow \Diamond q$, for some past formula p and q.

The example (continued) For the dining philosophers, the following properties are of interest:

- Mutual Exclusion[1]:
  $\forall ph \in \{1, \ldots, K\}.\Box\neg(< ph, _ > \in Eat$
  $\wedge < ph \oplus 1, _ > \in Eat)$
- Response:
  $\forall ph \in \{1, \ldots, K\}.\Diamond(< ph, _ > \in Eat)$

2.3 SAM models and their correctness

Formally, a software architecture model in SAM consists of a set of compositions $C = \{C_1, C_2, \ldots, C_k\}$ and a hierarchical mapping h relating compositions. Each composition $C_i = \{C_{m_i}, C_{n_i}, C_{S_i}\}$ consists of a set C_{m_i} of components, a set of C_{n_i} of connectors, and a set C_{S_i} of compositions constraints. An element $C_{i_j} = (S_{i_j}, B_{i_j})$, either a component or a connector, in a composition C_i has a property specification S_{i_j} (a temporal logic formula) and a behavior model B_{i_j} (a PrTN). Each composition constraint in C_{S_i} is also defined by a temporal logic formula. The interface of a behavior model B_{i_j} consists of a set of places (called *ports*) that is the interaction among relevant components and connectors. Each property specification S_{i_j} only uses the ports as its atomic predicates that are true in a given marking if they contain appropriate tokens. A composition constraint is defined as a property specification. However, it often contains ports belonging to multiple components and/or connectors. A component C_{i_j} can be refined into a lower-level composition C_ℓ, which is defined by $h(C_{i_j}) = C_\ell$.

The correctness of a SAM model is defined by the following criteria [7]:

- *Element Correctness*: The property specification S_{i_j} holds in the corresponding behavior model B_{i_j}, i.e. $B_{i_j} \models S_{i_j}$. Note we use B_{i_j} here to denote the set of behaviors or execution sequences defined by B_{i_j};
- *Composition Correctness*: The conjunction of all constraints in C_{S_i} of C_i is implied by the the conjunction of all the property specification S_{i_j} of C_{i_j}, i.e. $\bigwedge S_{i_j} \vdash \bigwedge C_{S_i}$. An alternative weaker but acceptable criterion is that the conjunction of all constraints in C_{S_i} holds in the integrated behavior model B_i of composition C_i, i.e. $B_i \models \bigwedge C_{S_i}$.

[1]The notation $< ph, _ >$ denotes a tuple, whose first element is ph and the remaining elements, if any, are not interesting.

- *Refinement Correctness*: The property specification S_{i_j} of a component C_{i_j} is implied by the composition constraints C_{s_ℓ} of its refinement C_ℓ with $C_\ell = h(C_{i_j})$, i.e. $\bigwedge C_{s_l} \vdash S_{i_j}$. An alternative weaker but acceptable criterion is that S_{i_j} holds in the integrated lower level behavior model B_ℓ of C_ℓ, i.e. $B_\ell \models S_{i_j}$.

These correctness criteria are the verification requirements of SAM models. In the next sections, we will show how element correctness and composition correctness are verified. Refinement correctness is out of the scope of this paper and will be treated elsewhere.

3 Formal analysis of SAM models

3.1 Fair transition systems

Our analysis method is based on STeP. In STeP, specifications are given in linear-time temporal logic as described in Section 2.2. Reactive systems are basically expressed by fair transition systems (FTS for short). We assume that all variables that describe states of programs or that appear in formulas specifying properties of programs are taken from the universal set of variables $\mathcal{V}$. For each $x \in \mathcal{V}$, its *primed version* x' is also in $\mathcal{V}$. To express the syntax of an FTS, we assume an underlying first-order language over $\mathcal{V}$. We define a *state s* to be an interpretation of $\mathcal{V}$, assigning to each variable $u \in \mathcal{V}$ a value $s[u]$ over its domain. We denote by Σ the set of all states.

An *FTS* $< V, \Theta, \mathcal{T}, \mathcal{J}, \mathcal{C} >$ is defined by the following components:

- $V = \{u_1, \ldots, u_n\} \subseteq \mathcal{V}$: A finite set of *system variables*.
- Θ: The *initial condition*.
- $\mathcal{T}$: A finite set of *transitions*. Each transition $\tau \in \mathcal{T}$ is a function
$$\tau : \Sigma \to 2^\Sigma,$$
mapping each state $s \in \Sigma$ into a (possibly empty) set of states $\tau(s) \subseteq \Sigma$. Each state in $\tau(s)$ is called a τ-*successor* of s.
We say that the transition τ is *enabled* on the state s if $\tau(s) \neq \emptyset$. Otherwise, we say that τ is *disabled* on s.
- $\mathcal{J} \subseteq \mathcal{T}$: A set of *just* transitions.
- $\mathcal{C} \subseteq \mathcal{T}$: A set of *compassionate* transitions.

Each transition $\tau \in \mathcal{T}$ is represented by a first-order formula $\rho_\tau(V, V')$, called the *transition relation*. Thus, the state s' is a τ-successor of the state s if the formula $\rho_\tau(V, V')$ evaluates to true. The enabledness of the transition τ can be expressed by the formula $En(\tau) : \exists V' : \rho_\tau(V, V')$.

Let $\mathcal{S}$ be an FTS for which the above components have been identified. We define a *computation* of $\mathcal{S}$ to be an infinite sequence of $\mathcal{V}$-states $\sigma : s_0, s_1, s_2, \ldots$, satisfying the following requirements:

- *Initiation:* s_0 is initial, i.e., $s_0 \models \Theta$.
- *Consecution:* For each $j = 0, 1, \ldots$, the state s_{j+1} is a τ-successor of the state s_j, i.e., $s_{j+1} \in \tau(s_j)$, for some $\tau \in \mathcal{T}$.
- *Justice:* For each $\tau \in \mathcal{J}$ it is not the case that τ is continually enabled beyond some point in σ but taken at only finitely many positions in σ.
- *Compassion:* For each $\tau \in \mathcal{C}$ it is not the case that τ is enabled on infinitely many states in σ but taken at only finitely many positions in σ.

For a system $\mathcal{S}$, we denote by $Comp(\mathcal{S})$ the set of all computation of $\mathcal{S}$.

3.2 From PrTNs to FTS models

Given a PrTN $\pi = (N, Spec, Ins)$ as defined in section 2.1, we define a corresponding FTS by $\mathcal{S}_\pi =< V, \Theta, \mathcal{T}, \mathcal{J}, \mathcal{C} >$, where

- The set of local variables: $V = \{v_p \mid p \in P\}$.
- Initial conditions:
$$\Theta = \bigwedge_{p \in P} (v_p = m_0(p))$$
- For every transition t in π and every α in the set of occurrence modes αS , there is a corresponding transition $\tau_{t,\alpha}$, whose transition relation is defined as $\rho_{\tau_{t,\alpha}} \stackrel{\text{def}}{=} (en(\tau_{t,\alpha}) \to f(\tau_{t,\alpha}))$, where
$$en(\tau_{t,\alpha}) = R(t) : \alpha \wedge \bigwedge_{p \in P} (v_p \supseteq L(p,t) : \alpha)$$
$$f(\tau_{t,\alpha}) = \bigwedge_{p \in P} (v'_p = v_p - L(p,t) : \alpha + L(t,p) : \alpha)$$
The set of all possible transitions $\mathcal{T} = \{\tau_{t,\alpha} \mid t \in T \wedge \alpha \in \alpha S\}$
- Weak fairness: $\mathcal{J} = \mathcal{T}$
- Strong fairness: $\mathcal{C} = \emptyset$

Reducing label substitutions There could be many redundant transitions in the resulting FTS, i.e. these transitions never occur due to the label constrains of a PrTN. Consequently, the label constraints can sometimes reduce the number of transitions in large scale. We usually apply the label constraints to reducing transitions before setting about formal specification and verification of the models.

Theorem 1 (Soundness of the transformation) *Given a PrTN π, suppose $\mathcal{S}_\pi =< V, \Theta, \mathcal{T}, \mathcal{J}, \mathcal{C} >$ be the FTS obtained from π using the above transformation rules.*

- *For any run $\sigma = m_0, m_1, m_2, \ldots \in Comp(\pi)$, $\forall v_p \in V, i \in N$, let $s_i(v_p) = m_i(p)$, then $\sigma' = s_0, s_1, s_2, \ldots \in Comp(S_\pi)$*

- *For any state sequence $\sigma = s_0, s_1, s_2, \ldots \in Comp(S_\pi)$, $\forall p \in P, i \in N$, let $m_i(p) = s_i(v_p)$, then $\sigma' = m_0, m_1, m_2, \ldots \in Comp(\pi)$*

Proof: (1) Suppose $\sigma = m_0, m_1, m_2, \ldots \in Comp(\pi)$. Let the corresponding transition sequence be $t_0 t_1 t_2 \ldots$, and occurrence models be $\alpha_0, \alpha_1, \alpha_2, \ldots$. By definition, the following predicate holds:

$$\forall i \in N, \forall p \in P.(m_i(p) \supseteq L(p,t) : \alpha_i) \wedge (R(t_i) : \alpha_i)$$
$$\wedge (m_{i+1} = m_i(p) - L(p,t) : \alpha_i + L(t,p) : \alpha_i)$$

Because $\forall v_p \in V, s_0(v_p) = m_0(p)$, we get $s_0 \models \Theta$. For each $i \in N$, $(s_i, s_{i+1}) \models \tau_{t_i,\alpha_i}$ holds. Consequently, $s_0 s_1 s_2 \ldots$ is a run of S_π and the corresponding transition sequence is $\tau_{t_0,\alpha_0}\tau_{t_1,\alpha_1}\tau_{t_2,\alpha_2}, \ldots$. Therefore $\sigma' = s_0, s_1, s_2, \ldots \in Comp(S_\pi)$.

(2) Suppose $\sigma = s_0, s_1, s_2, \ldots \in Comp(S_\pi)$. Let the corresponding transition sequence be $\tau_{t_0,\alpha_0}\tau_{t_1,\alpha_1}\tau_{t_2,\alpha_2}, \ldots$. So $m_i \in [m_0 >$ follows from the definition of m_i and τ_{t_i,α_i}. Consequently, $\sigma' = m_0, m_1, m_2, \ldots \in Comp(\pi)$. $\square$

The example (continued) Section 2.1 gives a PrTN model of the dining philosophers. By applying the above rules, we can obtain a corresponding FTS $\mathcal{S} =< V, \Theta, \mathcal{T}, \mathcal{J}, \mathcal{C} >$, where

- $V = \{think, chopstick, eat\}$

- $\Theta = (think = \{1, 2, \ldots, K\}) \wedge (eat = \{\ \})$
  $\wedge (chopstick = \{1, 2, \ldots, K\})$

- Using the reduction strategy for occurrence mode, we get

$$\alpha S = \{< ph, ch_1, ch_2 > \mapsto < 1, 1, 2 >,$$
$$< ph, ch_1, ch_2 > \mapsto < 2, 2, 3 >,$$
$$\ldots$$
$$< ph, ch_1, ch_2 > \mapsto < K, K, 1 >\}$$

$\mathcal{T} = \{\rho_{pickup} : \alpha, \rho_{putdown} : \alpha \mid \alpha \in \alpha S\}$, ρ_{pickup} is defined as $en(pickup) \rightarrow f(pickup)$ where:

$$en(pickup) = (think \supseteq \{ph\})$$
$$\wedge (chopstick \supseteq \{ch_1, ch_2\})$$
$$f(pickup) = (think' = think - \{ph\})$$
$$\wedge (chopstick' = chopstick - \{ch_1, ch_2\})$$
$$\wedge (eat' = eat \cup \{< ph, ch_1, ch_2 >\})$$

and $\rho_{putdown}$ is defined as $en(putdown) \rightarrow f(putdown)$ where:

$$en(putdown) = (eat \supseteq \{< ph, ch_1, ch_2 >\})$$
$$f(putdown) = (eat' = eat - \{< ph, ch_1, ch_2 >\})$$
$$\wedge (think' = think \cup \{ph\})$$
$$\wedge (chopstick' = chopstick \cup \{ch_1, ch_2\})$$

3.3 Verification method based on STeP

Our verification method is closely related to the hierarchical structure of SAM models. In the composition level, properties of components and connectors, as well as the entire system, are specified by temporal logic formulas. Deductive verification is most suitable for deriving properties of the entire system from the properties of its constituents. For element (either component or connector) analysis, we transform PrTNs into FTS models before STeP is resorted to.

A verification session in STeP begins by loading a program or transition system that describes the system of interest and entering a temporal-logic formula that expresses one or more properties to be proved. The formula becomes the root goal of a proof tree. Verification can be performed by the model checker or by deductive means.

The basic verification rule for safety properties is INV:

$$\frac{\Theta \rightarrow p, \quad \{p\}\mathcal{T}\{p\}}{\Box p}$$

By applying a suitable rule, a proof goal is reduced to a few first-order verification conditions or simpler temporal formulas, whose correctness is usually easier to establish.

4 Software architecture analysis: a case study

4.1 An electronic commerce system

The electronic commerce system (ECS) is a highly distributed WWW-based application [5]. In the ECS problem, there are customers and suppliers. Each customer has a contract with a supplier for purchases from that supplier, as well as one or more bank accounts through which payments to suppliers can be made. Each supplier provides a catalog of items, accepts customer orders, and maintains accounts with each customer for receiving payment. The ECS system can be conceptually illustrated

as a PrTN in Figure 2.

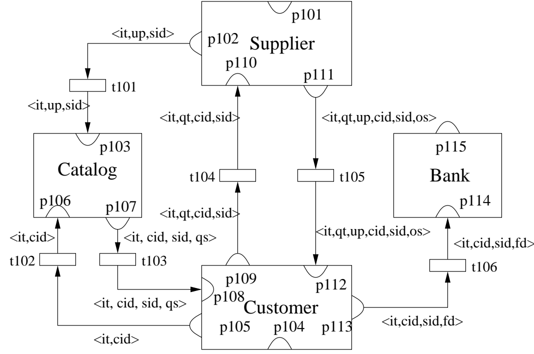

Figure 2. A PrTN model of the ECS

In the ECS problem, there are 4 components: *supplier, customer, catalog*, and *bank*. The interfaces of these components are a set of ports (places). There are 6 simple connectors, which connect these components. The arcs are labeled with variables or tuples of variables. The intuitive meaning of these variables are given in Table 1.

Table 1. Variables in Figure 2

Variable	*Description*
cid	Customer's identity
fd	Fund of a deal
it	Item number
os	Order status
qs	Query status
qt	Quantity of items
sid	Supplier's identity
up	Unit price

Each place has its type. The basic types include Int, Bool, Item, UnitPrice, SupplierID, CustomerID. Actually, the label of each arc indicates the type of the corresponding place. For example, the label of the arc from *p102* is $< it, up, sid >$, which indicates that the type of *p102* is $\wp(Item \times UnitPrice \times SupplierID)$. Places *p101, p104, p115* are used to denote the supplier's inventory, the customer's buy list, and the bank's transaction records, respectively.

4.2 Requirement specification

One requirement of the ECS system is:

If the supplier has products which meet the customer's needs, the deal will be made.

The requirement can be specified by the following temporal logic formula:

Req: $\forall it : \Box(< it, qt1, _ >\in p101 \wedge < it, qt2, _ >\in p104 \wedge qt1 \geq qt2 \rightarrow \Diamond(< it, _ >\in p115))$

The properties of components and connectors are also specified as temporal logic formulas.

(1) The supplier

- The supplier will advertise products in the catalog.
  S1: $\forall it : \Box(< it, _ >\in p101 \rightarrow \Diamond(< it, _ >\in p102))$
- The supplier will confirm the order and provide order status to the customer.
  S2: $\forall it : \Box(\Diamond(< it, qt1, _ >\in p101) \wedge \Diamond(< it, qt2, _ >\in p110) \wedge qt1 \geq qt2 \rightarrow \Diamond(< it, _, true >\in p111))$

(2) The customer

- The customer will submit request to catalog for the interested products.
  C1: $\forall it : \Box(< it, _ >\in p104 \rightarrow \Diamond(< it, _ >\in p105))$
- The customer will order the product if it satisfies his needs.
  C2: $\forall it : \Box(\Diamond(< it, qt, _ >\in p104) \wedge \Diamond(< it, _, true >\in p108) \rightarrow \Diamond(< it, qt, _ >\in p109))$
- Once the merchandise is received, the customer will confirm the payment.
  C3: $\forall it : \Box(< it, _, true >\in p112 \rightarrow \Diamond(< it, _ >\in p113))$

(3) The catalog

- Customer's query will be properly answered.
  Ca: $\forall it : \Box(\Diamond < it, _ >\in p103 \wedge \Diamond < it, _ >\in p106 \rightarrow \Diamond(< it, _, true >\in p107))$

(4) The bank

- Operation funds will be transferred upon customer's confirmation.
  Ba: $\forall it : \Box(< it, _ >\in p114 \rightarrow \Diamond(< it, _ >\in p115))$

(5) The connectors: Each connector acts simply as a pipe.

- T101: $\forall it : \Box(< it, _ >\in p102 \rightarrow \Diamond(< it, _ >\in p103))$
- T102: $\forall it : \Box(< it, _ >\in p105 \rightarrow \Diamond(< it, _ >\in p106))$
- T103: $\forall it : \Box(< it, _ >\in p107 \rightarrow \Diamond(< it, _ >\in p108))$
- T104: $\forall it : \Box(< it, _ >\in p109 \rightarrow \Diamond(< it, _ >\in p110))$
- T105: $\forall it : \Box(< it, _ >\in p111 \rightarrow \Diamond(< it, _ >\in p112))$
- T106: $\forall it : \Box(< it, _ >\in p113 \rightarrow \Diamond(< it, _ >\in p114))$

4.3 Refinement and analysis of components

Figure 3 shows a PrTN model of one component, the supplier, which is a refinement of that in Figure 2.

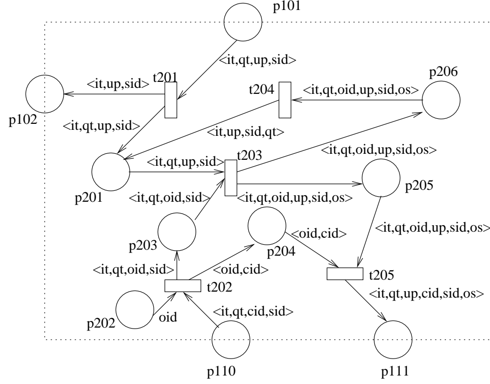

Figure 3. The supplier

The only local variable in Figure 3 which did not appear before is *oid*, who acts the role of order tracking number. The intuitive functionality of transitions is given in Table 2.

Table 2. Transitions in Figure 3

Transition	*Functionality*
t201	Adervertizing products
t202	Adding an ID number to each order
t203	Checking the inventory
t204	Maintaining the inventory
t205	Delivering the order

Using the transformation rules in Section 3.2, the PrTN model can systematically transformed into an FTS model.

Suppose there are three kinds of items in the inventory and two order requests submitted. Properties of the supplier coded in STeP like following:

```
SPEC
 PROPERTY S1:  Forall i:[1..3].
   (♯2 p101[i]>0
      --> <>(♯ 2 p102[i]= true))
 PROPERTY S2:  Forall i:[1..2].
   ((<>(♯2 p110[i]>0)
   /\ <>(♯2 p110[i]>0)
   /\ (♯2 p110[i]<= ♯2 p101[i]) )
      --> <> (♯2 p111[i]> 0
               /\ ♯3 p111[i]= true))
```

After loading both of the FTS model and the property specifications, we are ready to verify the correctness of the system. Because the system is finite state, both of the above properties are easily established using "ModelChecker" of STeP. The other models of components are illustrated in Figure 4-6. Their specification and verification are similar to those of the Supplier and omitted.

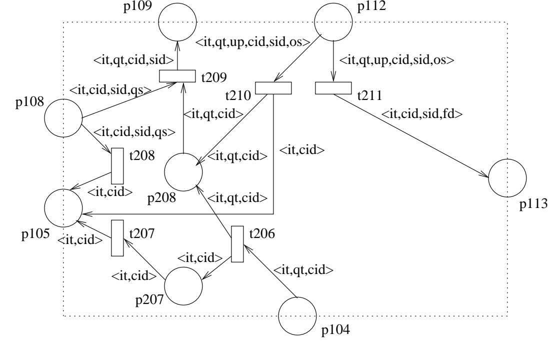

Figure 4. The customer

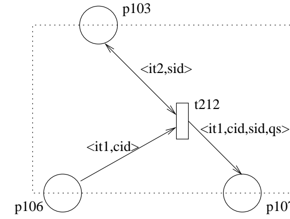

Figure 5. The catalog

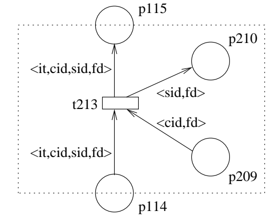

Figure 6. The bank

4.4 Analysis of composition correctness

Composition correctness of the ECS is ensured by the constraints of the components and connectors. Formally, Let $\Gamma \overset{\text{def}}{=} \{$S1, S2, C1, C2, C3, Ca, Ba, T101, T102, T103, T104, T105, T106 $\}$. We can show $\Gamma \vdash Req$. The following is a proof outline.

(1) $< it, qt_1, _ > \in p101 \rightarrow \Diamond < it, _ > \in p102$ (S1)

(2) $< it, _ > \in p102 \rightarrow \Diamond < it, _ > \in p103$ (T101)

(3) $< it, qt_1, _ > \in p101 \rightarrow \Diamond < it, _ > \in p103$ (TL(1),(2)) [2]

(4) $< it, qt_2, _ > \in p104 \rightarrow \Diamond < it, _ > \in p105$ (C1)

(5) $< it, _ > \in p105 \rightarrow \Diamond < it, _ > \in p106$ (T102)

(6) $< it, qt_2, _ > \in p104 \rightarrow \Diamond < it, _ > \in p106$ (TL(4),(5))

(7) $< it, qt_1, _ > \in p101 \wedge < it, qt_2, _ > \in p104$
$\rightarrow \Diamond < it, _ > \in p103 \wedge \Diamond < it, _ > \in p106$ (TL(3),(6))

(8) $\Diamond < it, _ > \in p103 \wedge \Diamond < it, _ > \in p106$
$\rightarrow \Diamond < it, _, true > \in p107$ (Ca)

(9) $< it, qt_1, _ > \in p101 \wedge < it, qt_2, _ > \in p104$
$\rightarrow \Diamond < it, _, true > \in p107$ (TL(7),(8))

(10) $< it, _ > \in p107 \rightarrow \Diamond < it, _ > \in p108$ (T103)

(11) $< it, qt_1, _ > \in p101 \wedge < it, qt_2, _ > \in p104$
$\rightarrow \Diamond < it, _, true > \in p108$ (TL(9),(10))

(12) $\Diamond < it, _, true > \in p104 \wedge$
$\Diamond < it, qt_2, _ > \in p108$
$\rightarrow \Diamond < it, qt_2, _ > \in p109$ (C2)

(13) $< it, qt_1, _ > \in p101 \wedge < it, qt_2, _ > \in p104$
$\rightarrow \Diamond < it, qt_2, _ > \in p109$ (TL(11),(12))

(14) $< it, qt_2, _ > \in p109 \rightarrow \Diamond < it, qt_2, _ > \in p110$ (T104)

[2] The notation "TL(1),(2)" denotes that the current formula is derived from formulas (1) and (2) using temporal logic proof rules.

(15)$< it, qt_1, _ > \in p101 \wedge < it, qt_2, _ > \in p104$
$\rightarrow \Diamond < it, qt_2, _ > \in p110$ (TL(13),(14))
(16) $\Diamond < it, qt_1, _ > \in p101 \wedge$
$\Diamond < it, qt_2, _ > \in p110 \wedge qt_1 \geq qt_2$
$\rightarrow \Diamond < it, qt_2, _, true > \in p111$ (S2)
(17) $< it, qt_1, _ > \in p101 \wedge$
$< it, qt_2, _ > \in p104 \wedge qt_1 \geq qt_2$
$\rightarrow \Diamond < it, qt_2, _, true > \in p111$ (TL(15),(16))
(18) $< it, qt_2, _, true > \in p111$
$\rightarrow \Diamond < it, qt_2, _, true > \in p112$ (T105)
(19) $< it, qt_1, _ > \in p101 \wedge$
$< it, qt_2, _ > \in p104 \wedge qt_1 \geq qt_2$
$\rightarrow \Diamond < it, qt_2, _, true > \in p112$ (TL(17),(18))
(20) $< it, qt_2, _, true > \in p112$
$\rightarrow \Diamond < it, _ > \in p113$ (C3)
(21) $< it, qt_1, _ > \in p101 \wedge$
$< it, qt_2, _ > \in p104 \wedge qt_1 \geq qt_2$
$\rightarrow \Diamond < it, _ > \in p113$ (TL(19),(20))
(22) $< it, _ > \in p113 \rightarrow \Diamond < it, _ > \in p114$ (T106)
(23) $< it, qt_1, _ > \in p101 \wedge$
$< it, qt_2, _ > \in p104 \wedge qt_1 \geq qt_2$
$\rightarrow \Diamond < it, _ > \in p114$ (TL(21),(22))
(24) $< it, _ > \in p114 \rightarrow \Diamond < it, _ > \in p115$ (Ba)
(25) $< it, qt_1, _ > \in p101 \wedge$
$< it, qt_2, _ > \in p104 \wedge qt_1 \geq qt_2$
$\rightarrow \Diamond < it, _ > \in p115$ (TL(23),(24))

Formula (25) is exactly the requirement of the ECS example we want to establish. As we can see, the entire proof is quite straightforward. We mainly use the temporal rules 'chain' and 'modus ponens' in the proof.

5 Conclusion

This paper proposes a formal method for analyzing software architecture models in SAM. The properties of individual components and connectors are verified either by deductive reasoning or model checking. The properties of the entire system is inferred from the properties of its constituents by deductive reasoning. Our verification method is based on the STeP. In contrast to SMV [11], STeP supports first-order reasoning in addition to model checking, which fits our needs well. However, we only used 'ModelChecker' of STeP in this case, for STeP does not directly support temporal rules such as 'chain'. Our method differs from the modular proof proposed in [2]. Although the modular proof has some advantages, it depends on the parallel combination of modules, which can lead to state explosion.

Software architecture and architecture description languages (ADLs) have recently become areas of intense research in software engineering society [13]. Architectural descriptions are often intended to model large, distributed, concurrent systems. The ability to evaluate the properties of such systems upstream can substantially lessen the cost of any errors. Architecture level analysis can benefits from the special characteristics of ADLs, which will hopefully make the task easier [12]. Extraction of these characteristics of ADLs, especially for SAM, is one of our future research interests.

References

[1] K. Apt and E. Olderog. *Verification of Sequential and Concurrent Programs*. Springer-Verlag, 1991.
[2] N. Bjørner, Z. Manna, H. Sipma, and T. Uribe. Deductive verification of real-time systems using STeP. *Theoretical Computer Science*, 253:27–60, 2001.
[3] E. Clarke and J. Wing. Formal methods: state of the art and future directions. *ACM Computing Surveys*, 28(4):626–643, 1996.
[4] H. Genrich and K. Lautenbach. System modeling with high-level Petri nets. *Theoretical Computer Sicence*, 13:109–136, 1981.
[5] H. Gomma. *Design Concurrent, Distributed, and Real-Time Applications with UML*. Addison-Wesley, 2000.
[6] X. He. A formal definition of hierarchical predicate transition nets. In *Proceedings of the 17th International Conference on Application and Theory of Petri Nets, LNCS 1091*, pages 212–229. Springer-Verlag, 1996.
[7] X. He and Y. Deng. A framework for developing and analyzing software architecture specifications in SAM. *The Computer Journal*, (accepted), 2002.
[8] Z. Manna, A. Anuchitanukul, N. Bjørner, A. Browne, E. Chang, M. Colón, L. de Alaro, H. Devarajan, H. Sipma, and M. Uribe. STeP: the Stanford Temporal Prover. Technical Report STAN-CS-TR-94-1518, Department of Computer Science, Stanford University, June 1994.
[9] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, 1992.
[10] Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety*. Springer-Verlag, 1995.
[11] K. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.
[12] N. Medvidovic and R. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 26(1):70–93, 2000.
[13] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall, Inc., 1996.
[14] J. Wang, X. He, and Y. Deng. Introducing software architecture specification and analysis in SAM through an example. *Information and Software Technology*, 41:451–467, 1999.