Modeling and Analyzing Software Architecture Using Object-Oriented Petri Nets and π-calculus

Zhenhua Yu[1,2], Xiao Fu[1], Yu Liu[2], Jing Wang[2] and Yuanli Cai[3]
[1]*School of Telecommunication Engineering, Air Force Engineering University*
[2] *Xi'an Applied Optics Institute*
[3]*School of Electronic and Information Engineering, Xi'an Jiaotong University*
People's Republic of China

1. Introduction

Software architecture has recently emerged as a new discipline of software engineering to effectively develop and maintain complex and large-scale software systems and reduce costs of developing applications. Software architecture provides a high-level abstraction for representing components, their relationships to each other and environment, and their constraints. The overall system structure design and specifications are far more important than the selection of the specific algorithms and data structures. Therefore, software architecture is a critical factor to success for system design and development (Shaw & Clements, 2006).
Software architecture can be characterized according to its evolution at runtime (Oquendo, 2004): 1. static architectures: the architecture does not evolve during the execution of the system; 2. dynamic architectures: the architecture can evolve during the execution, e.g. components can be created, deleted, reconfigured, or updated at run-time. Dynamic software architectures have several practical applications (Medvidovic & Taylor, 2000). In public information systems with high availability and in mission- and safety-critical systems, the implementation of architectural evolvement at run-time can decrease the cost and risk. To support architecture-based development, architecture description languages (ADLs) and formal models have been proposed to represent software architecture in a formal way, such as UniCon (Shaw et al., 1995), Darwin (Magee, 1995), Rapide (Luckham et al., 1995), Wright (Allen et al., 1998), π-ADL (Oquendo, 2004), SAM (He et al., 2004), XYZ/ADL (Luo et al., 2000). However, the major attentions have been focused on the description of static architectures, while the description of dynamic architecture has not yet to receive the attention it deserves. Darwin and Rapide only depict predefined dynamic evolvement and cannot verify the integrality and liveness of the systems. Wright can describe the dynamic evolvement, but it is so complicated. For a two-tier client/server system, process algebra (such as π-calculus) uses two processes and one or two channels to depict it, while Wright only needs seven processes and eight channels. π-SPACE and π-ADL cannot analyze the key characteristics. Although the existing approaches provide support for dynamic software architecture, most of them cannot analyze and verify the key characteristics. Therefore, software systems cannot be ensured robustness, consistency and maintenance.
To support the development of correct and robust dynamic software architectures, a visual software architecture formal model (SAFM) based on two complementary formalisms, namely Object-oriented Petri nets (OPN) and π-calculus, is proposed. SAFM divides software systems

into components, connectors and configuration module. In SAFM, OPN are employed to visualize the static architecture and depict the behavior of software systems, and π-calculus is used to describe software architecture evolution, including component joining, exiting, updating, load balancing and architecture reconfiguration. As π-calculus, which is based on the interleaving semantics, cannot depict the true concurrency and has few supporting tools, the π-calculus model of architecture evolution is translated into Petri nets (Yu et al., 2007). Consequently the structural analysis techniques allow the qualitative analysis of properties that may be proved directly on the structure of Petri nets, and the final model can be directly analyzed and verified using existing Petri net tools. SAFM approach supports detection of design errors in an early software design stage and the quality of the software can be significantly improved.

2. Object-oriented Petri Nets, π-calculus and Their Integration

2.1 A New Object-oriented Petri Nets (OPN)

The ordinary Petri nets models are very complicated, which highly depend on the system and lack the modularity and flexibility. Consequently state explosion in ordinary Petri net modeling is easily occurred. To solve the complexity and state explosion, Petri nets are combined with Object-oriented methods to set up Object-oriented Petri nets. Object-oriented Petri nets can tersely and independently represent all kinds of resources in a complex system, increase the flexibility of the model. Many kinds of Object-oriented Petri nets (Miyamoto & Kumagai, 2005) are presented. However many of them cannot completely describe the characteristics of objects. From software components perspective, a new Object-oriented Petri nets (OPN) are presented. In OPN both the modularity and flexibility are better than those of ordinary Petri nets, and the state explosion problem is a little more alleviated.
The OPN model of a physical object is defined as follows.
Definition 1 OPN is a 9-tuple, $OPN = (\Sigma, P, T, IT, OT, F, E, G, C)$, where Σ is color sets, which is a finite set of data types, variables and functions; where P is a finite set of places, $P = \{p_1, p_2, \ldots, p_j\}$; T is a finite set of transitions, $T = \{t_1, t_2, \ldots, t_k\}$; IT (Input Transition) and OT (Output Transition) are sets of input and output transitions, $IT = \{it_1, it_2, \ldots, it_l\}$, $OT = \{ot_1, ot_2, \ldots, ot_m\}$; $F \subseteq (P \times T) \bigcup (T \times P) \bigcup (P \times IT) \bigcup (IT \times P) \bigcup (P \times OT) \bigcup (OT \times P)$ is the input and output relationships between transitions and places; $E : F \rightarrow (ID, CDS)$ is expression functions in the arcs, ID is the identification of the arc and $CDS \subseteq \Sigma$ is a complicated data structure; G is the guard function of the transitions, which is a boolean expression. $C(P) \subseteq \Sigma$ is a set of color associated with the places P.
A system is composed of objects and their interconnection relations, and its formal definition is given as follows.
Definition 2 A system is a 3-tuple, $S = (OPN, Gate, C)$, where OPN is a finite set of physical objects in the system, $O = \{OPN_1, OPN_2, \ldots, OPN_i\}$; $Gate$ is a finite set of communication places, which are message passing relations among OPN; $C(Gate) \subseteq \Sigma$ is a set of color associated with the places $Gate$.
OPN can represent the object-oriented characteristics, such as encapsulation, inheritance and polymorphism. The behavior equivalence of models can be judged by the branch bisimilarity (Yu, 2006)

2.2 π-Calculus

The π-calculus (Milner et al., 1992) is an extension of the process algebra CCS (Calculus of Communicating Systems) in order to allow dynamic reconfiguration of systems. The model-

ing entities are *names* and *processes*. Systems are represented as a set of *processes* which interact by means of *names*. The *names* can be regarded as shared channels, variables or constants, which act as subjects for interaction. The *process* can use the *name* as a subject for future transmissions, which allows an easy and effective reconfiguration of the system.
We assume an infinite set of *names* $\mathcal{N}$, ranged over by *a, b, ..., z*, which will function as all of channels, variables and data values; a set of *process identifiers* $\mathcal{K}$ is ranged over by $A, B, \ldots$, each with an arity (an integer ≥ 0); the *processes* are ranged over by *P, Q, R, ...*, which are of seven kinds as follows:

1. A *Sum* $\sum_{i \in I} P_i$ representing the process that can enact one or other of the P_i.
2. A *prefix* form $\overline{y}x.P$, $y(x).P$, or $\tau.P$.

   $\overline{y}x.$ is called *negative* prefix. $\overline{y}$ may be thought of as an *output port* of a process; $\overline{y}x.P$ outputs the name x at port $\overline{y}$ and then behaves like P.

   $y(x).$ is called *positive* prefix. y may be thought of as an *input port* of a process; $y(x).P$ inputs an arbitrary name z at port y and then behaves like $P\{z/x\}$.

   $\tau.$ is called *silent prefix*, which represents an agent that can evolve to P without interaction with environment. $\tau.P$ performs the *silent action* τ and then behaves like P.
3. A *Parallel Composition* $P|Q$, which represents the combined behaviors of P and Q executing in parallel. The processes P and Q can act independently, and may also communicate if one performs an output and the other an input along the same port.
4. A *restriction* $(\nu x)P$. This process behaves as P but the name x is local, meaning it cannot immediately be used as a port for communication between P and its environment.
5. A *match* $[x = y]P$. This process behaves like P if the names x and y are identical, otherwise it does nothing.
6. A *defined agent* $A(y_1, \ldots, y_n)$. For any process identifier A (with arity n) used thus, there must be a unique *defining equation* $A(x_1, \ldots, x_n) \overset{\text{def}}{=} P$, where the names $x_1, \ldots, x_n$ are distinct and are the only names which may occur free in P.
7. A *Replication* $!P$. $!P$ is given by the definition $!P \overset{\text{def}}{=} P|!P$, which represents an unbounded number of copies of P.

The π-calculus can be varied in many ways. There are many useful subcalculi, e.g. the polyadic π-calculus (Milner, 1993). The polyadic π-calculus allows multiple objects in communications: outputs of type $\overline{a}\langle y_1, \ldots, y_n \rangle.P$ and inputs of type $a(x_1, \ldots, x_n).Q$. In this paper, the polyadic π-calculus is adopted as the modeling tool.
π-calculus can address the description of system with a dynamic or evolving topology, and analyze the key properties, such as deadlock, bisimulation, and bisimilarity.

2.3 The Integration of Petri Nets and π-calculus

Petri nets are graphical representation and a promising tool to describe the static characteristics of the system, represent the dynamic behaviors, and express causality and concurrency in system behavior. Structural properties of Petri nets, such as P-invariants and T-invariants, are employed to analyze the relations of the structure and behaviors of a system. Furthermore, Petri nets provide a variety of well-established mathematical methods to analyze, simulate and validate the systems. These properties make Petri nets as an excellent tool for the validation of models by non-technical end users. However the structure of Petri nets is static, it is hardly possible to model dynamic system architecture.

π-calculus is suitable for describing software system with an evolving communication topology. π-calculus can specify and reason about the design of complex concurrent computing systems by means of algebraic operators corresponding to common programming constructs Best et al. (2001). However,the processes of π-calculus are complicated, and it cannot visually model the system architecture (Jiang, 2003). Moreover, as π-calculus, which is based on the interleaving semantics, cannot depict the true concurrency and has few supporting tools.

The treatment of the structure and semantics of concurrent systems provided by Petri nets and π-calculus is different, so it is virtually impossible to take full advantage of their overall strengths when they are used separately. Therefore the idea of combining Petri nets and π-calculus is proposed, where Petri nets are employed to visually model the system architecture and system behaviors, and π-calculus is employed to describe the system evolution. To remedy the deficiencies of π-calculus, π-calculus is mapped into Petri nets to visualize system structure as well as system behaviors. Therefore, the structural analysis techniques allow direct qualitative analysis is of the system properties on the structure of the nets.

The use of dual complimentary formal methods has many advantages over a single formalism (Clarke, 1996), including modeling and analyzing different aspects of software architecture using different formalism to improve understandability. The integration of Petri nets and π-calculus provides a bridge between graphical specification techniques and dynamic modeling techniques. π-calculus and Petri nets can complement each other very well.

3. Software Architecture Formal Model

A visual software architecture formal model (SAFM) based on Object-oriented Petri nets and π-calculus, is proposed. SAFM models and analyzes software architecture, and it describes the components, connectors and configuration.

Definition 3 SAFM is a three-tuple, $SAFM = (Comp, Conn, Conf)$, where $Comp = (Comp_1, Comp_2, \ldots, Comp_o,)$ is a set of components, $Conn = (Conn_1, Conn_2, \ldots, Conn_p)$ is a set of connectors, and $Conf$ is architecture configuration.

3.1 Modeling Components

A component is a unit of data or computation, loci of status store and computation with extended and integrated. A Component is 3-tuple, $Comp_o = (ID, OPN, \Pi)$, where ID is the identifier of a component; OPN defines the interfaces and internal implementation of a component; Π describes the evolvement of a component by π-calculus.

In OPN, IT and OT describe a component's interfaces that are a set of interaction points between it and the external world, $Comp_o.Interface = \{(t_1, t_2) | t_1 \in IT, t_2 \in OT\}$. The interface specifies the services a component requires and provides, especially the messages a component receives and sends. The implementation of a component is described by other tuples of OPN. A component interacts with other components by interfaces, and its internal implementation is invisible for other components. The evolve process of components will be described in the next section.

Components are reusable software units, including composite components and atomic components. A composite component may be composed of other composite components or atomic components. An atomic component is no longer divided.

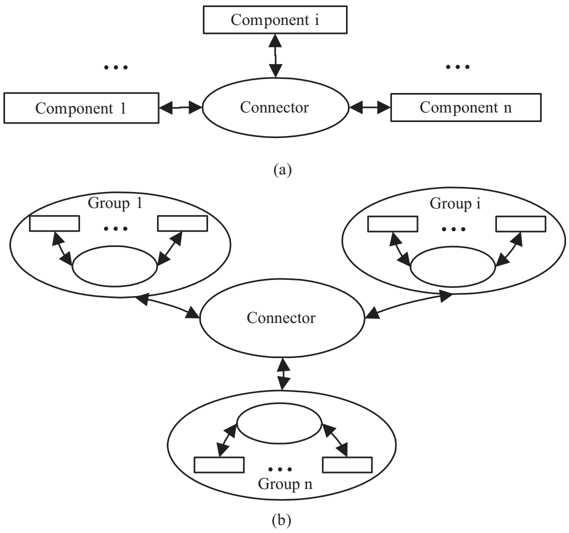

Fig. 1. The connector model

3.2 Modeling Connectors

Connectors are used to model the interactions among components, define the rules that govern those interactions. Connectors coordinate and supervise components from the high level, and manage the resources of the system.

A connector is defined as $Conn_p = (ILP, Gate, KBP, Role, \Pi)$, where ILP is a intelligent link place denoted by a ellipse. The information obtained from the external is saved in the ILP to set up message passing channels among components. *Gate* is the tuple in OPN model. *KBP* represents Knowledge-base Place which is defined to apperceive the external environment, acquire requisite knowledge, and describe services which components provide via interfaces. *Role* is a set of components interact with the connector, which is defined as $Role = \{CID_1, \ldots, CID_n\}$. Π addresses the evolvement of connectors by π-calculus, which will be described in the next section.

From the point of view of communication, the connector controls and manages the communication and collaboration among components; from the point of view of the system connection and conglutination, the connector plays the role of the glue conglutinating the software system.

In the connector, the roles identify the logical participants in the interaction. There are two types of roles, static and dynamic role, respectively. Dynamic role will change with the components deleted or added.

A software system may consist of some connectors. If a system is composed of a connector and some components to achieve a certain goal, then it is called a group, which is shown in Fig. 1(a). Fig. 1(b) represents several groups constitute a large-scale system, and these groups is connected by a connector.

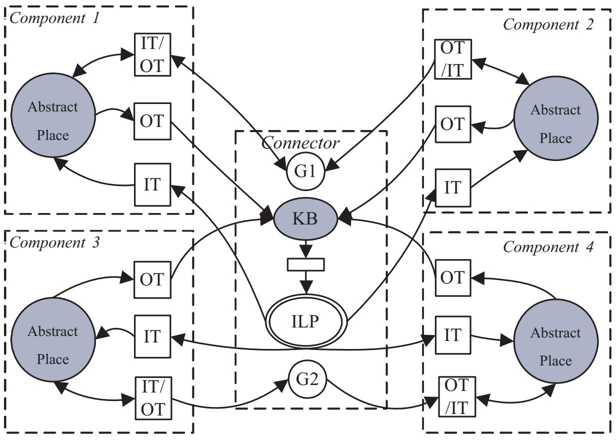

Fig. 2. software architecture configuration

3.3 Modeling Configurations

$Conf$ is the architectural configuration, which addresses the connected graphs of components and connectors. $Conf$ is studied from the macro-level, where software systems are conceived as a multitude of interacting components and connectors. In the macro-level, the key point is the overall structure and behaviors, rather than the mere behaviors of individuals.
The architectural configuration is shown in Fig. 2. For simplicity and clarity of the diagrams, this configuration model is predigested. The components are represented by IT, OT and abstract places denoted by shaded circles. The abstract places can be refined according to requirements. The static semantics of architecture is visually described in Fig. 2, and the dynamic semantics is represented by the firing of transitions. The firing of the transition makes the Token dispatch, which expresses the message passing and well depicts interactions among components.

4. The Dynamic Evolvement of Software Architecture

To address dynamic software architecture, the scheme in SAFM is defined as follows.

1. The supervising processes are defined in components and connectors. The supervising processes in components send messages to connectors, and describe the internal reason of dynamic evolvement, such as computation errors or abnormity. The supervising processes in connectors interact with the supervising processes in components and environment, and describe the exterior reason of dynamic evolvement.
2. The connectors is the supervisor of a system, and defined the operators, such as $Create, Delete, Update$, to describe the dynamic evolvement.
3. After addressing the dynamic evolvement, the correctness and consistency must be analyzed, which will be described in the next section.

The supervising processes in components can be defined as follows.

$$Monitor(request, config) = \overline{request}\langle id, updinfo\rangle.config(x,y)([x = id, y = begin]Update).$$
$$Reginfo\langle id, s\rangle.Monitor(request, config) \quad (1)$$

where the process $Monitor$ sends id and update information $updinfo$ to the supervising process in a connector via the channel $request$, and then waits for the notice to update. After the component updates using the process $update$, the process $Reginfo$ will register its related information in the connector.
The supervising processes in connectors can be defined as follows.

$$Supervisor(request, info, config) = request(u, v).\overline{info}\langle ids, wait\rangle.\overline{config}\langle id, begin\rangle. \\ CReginfo\langle x, y\rangle.Supervisor(request, info, config) \quad (2)$$

where the process $supervisor$ receives the component updating information via the channel $request$, and notifies the related components suspend their services via the channel $info$, and then accepts the enrollment information of the new component.

4.1 Components Joining and Exiting

In software systems, new components first enroll their information (such as name, address, interface and capability) in connectors, and set up the channels for interacting with the other components via ILP. The creating process of a component is

$$NewComp(id, s) = Create(id, s) \quad (3)$$

It means that a component is created with a identifier id and providing a service s. The enroll process of a component is

$$RegInfo(id, s) = (\nu id, s)(\overline{register}\langle id, s\rangle) \quad (4)$$

It means that a component enrolls a service s and its identifier id via the channel $register$ to a connector, and the id, s is private names. The corresponding enroll process in the connector is

$$CRegInfo(x, y) = (\nu x, y)(register(x, y)) \quad (5)$$

It means that the connector obtains a service information via the channel $register$, and the x, y is private names.
When a component requests a service, the connector queries the knowledge base to search a corresponding component providing the service to send its identifier to the requesting component. If the requesting component receives the identifier of the service component, it sends the message to the service component through the connector; if the service component does not exist, the requesting component can subscribe for this service. The connector will inform the requesting component as long as it becomes aware of the information that a corresponding component registers. This requesting process in the requesting component is

$$RequestService(i, r, l) = \bar{i}\langle a\rangle.r(z).([z = nil].\overline{subscribe}\langle a\rangle + \bar{z}\langle l\rangle) \quad (6)$$

The requesting component sends the request a through the channel i to the connector to query the corresponding service component, and then wait a response from the connector through the channel r. After the requesting component receives the identifier of the service component z, it sends the requesting address l to the service component by the channel z.
The service query process in the connector is

$$QueryService(i, r, p) = i(y).(\overline{kb}\langle y\rangle|Belief(y)).(\bar{r}\langle nil\rangle.subscribe(y) + \bar{r}\langle P\rangle) \quad (7)$$

and the corresponding process in the service component is

$$ProvdService(p,s) = p(x).\bar{x}\langle s\rangle \tag{8}$$

The service component sends the service through the channel x to the requesting component. According to the above analysis, the entire dynamic process of the service requesting and providing is modeled as follows:

$$\begin{aligned}
&RequestService(i,r,l)|QueryService(i,r,p)|ProvdService(p,s)\\
&= \bar{i}\langle a\rangle.r(z).([z = nil].\overline{subscribe}\langle a\rangle + \bar{z}\langle l\rangle)|i(y).(\overline{kb}\langle y\rangle|Belief(y)).(\bar{r}\langle nil\rangle.subscribe(y) +\\
&\bar{r}\langle P\rangle)|p(x).\bar{x}\langle s\rangle\\
&\xrightarrow{\tau} r(z).([z = nil].\overline{subscribe}\langle a\rangle + \bar{z}\langle l\rangle)|(\overline{kb}\langle a\rangle|Belief(a)).(\bar{r}\langle nil\rangle.subscribe(y) +\\
&\bar{r}\langle P\rangle)|p(x).\bar{x}\langle s\rangle\\
&\xrightarrow{\tau} r(z).([z = nil].\overline{subscribe}\langle a\rangle + \bar{z}\langle l\rangle)|\bar{r}\langle P\rangle|p(x).\bar{x}\langle s\rangle\\
&\xrightarrow{\tau} \bar{p}\langle l\rangle|p(x).\bar{x}\langle s\rangle\\
&\xrightarrow{\tau} \bar{l}\langle s\rangle
\end{aligned}$$

If a component achieves its goal and wants to exit from the system, it must delete its information, so the information in the connector will not fall into confusion. The exiting process of a component is

$$Comp(id,s) = Delete(id,s) \tag{9}$$

It means that a component is deleted with an identifier *id* and a service s.

4.2 Component Update

Component update can be classified into two categories. 1. The algorithm of a component may be error or its deficiency is lower, so the component must be updated, which is called holding semantic update. 2. For new system requirements appearing, a component with new functions will update the former component, which is called extended update.

For the first case, the weak equivalent of π-calculus can be used to judge whether the new component substitutes for the old component. If the behaviors of two components are equivalent, a component can substitute for the other one, which the environment cannot apperceive. The holding semantic update is defined as follows.

Rule 1 Suppose the behaviors of the components $Comp_0$ and $Comp_1$ are the processes P and Q, respectively. If $P \approx Q$, $Comp_1$ can update $Comp_0$ denoted as $P \rhd Q$, which is called holding semantic update.

Holding semantic update means that the internal algorithms of components are updated, while their behaviors are not changed.

For the second case, a powerful component updates the former component. The extended update is defined as follows.

Rule 2 Suppose the behaviors of the components $Comp_0$ and $Comp_1$ are the processes P and Q, respectively. P and Q satisfy the following conditions.

1. $fn(P) \subseteq fn(Q)$;
2. If $P \xrightarrow{\tau} P'$, then $\exists Q', Q \Rightarrow Q'$;
3. If $P \xrightarrow{x(z)} P'$, then $\exists Q', Q \xrightarrow{x(z)...x_i(z_i)} Q'$($\xrightarrow{x(z)...x_i(z_i)}$means that Q can execute the other actions $x_i(z_i)$ except $x(z)$);

4. If $P \xrightarrow{\overline{x}\langle y\rangle} P'$, then $\exists Q', Q \xrightarrow{\overline{x}\langle y\rangle ... \overline{x_i}\langle y_i\rangle} Q'$($\xrightarrow{\overline{x}\langle y\rangle ... \overline{x_i}\langle y_i\rangle}$means that Q can execute the other actions $\overline{x_i}\langle y_i\rangle$ except $\overline{x}\langle y\rangle$);
5. If $P \xrightarrow{\overline{x}y} P'$, then $\exists Q', Q \xrightarrow{\overline{x}y ... \overline{x_i}y_i} Q'$.

Then Q updates P, therefore $Comp_1$ can update $Comp_0$ denoted as $Comp_1 \triangleright \triangleright Comp_0$, which is called extended update.
Extended update means that the new component provides new functions except remaining the former behaviors.

4.3 Load Balancing

In distributed systems, if a new server is added, the requests of clients must be assigned to different servers to balance load. In SAFM, connectors are employed to balance load of servers. For example, if a system consists of two server components, load balancing rule is

$$\begin{aligned} &\text{if}(Server_1.cn < Server_2.cn) \\ &\text{then } QueryService(i_1, r_1, p_1) \\ &\text{else } QueryService(i_2, r_2, p_2) \end{aligned}$$

It means that before the connector sets up a channel for a client and a server, it must judge the number of clients interacting with the two servers. If the number of clients interacting with $Server_1$ is less than that of clients interacting with $Server_2$, the identifier of $Server_1$ is transmitted to the clients.

4.4 Architecture Reconfiguration

To improve the stability of systems, some backup components are added. When the primary component goes down, the backup component is used until the primary component returns to service. In Fig. 2, $component_1$ interacts with $component_2$ via the channel G_1 (denoted as y). During the system running, a backup component $component_{1'}$ is added to backup the data via the channel bak. If $component_1$ goes down, $component_2$ needs switch its channel to $component_{1'}$. The dynamic configuration process is shown as follows.

$$\begin{aligned} &(\nu bak)(\overline{y}\langle bak\rangle.Comp_1|Comp_{1'})|y(x).Comp_2 \\ &\xrightarrow{\tau} (\nu bak)(Comp_1|Comp_{1'})|Comp_2\{bak/x\} \end{aligned}$$

It means that before $component_1$ going down, the private channel bak is transferred to $component_2$. Therefore $component_2$ can interact with $component_1$ via the channel bak.

5. Analyzing Software Architecture

For analyzing static software architecture, the related methods and supporting tool (such as INA (Roch & Starke, 2009)) can be employed to analyze deadlock, boundness and reachbility of models. In this section, the major attentions will be focused on analyzing dynamic software architecture. Owing to components interacting with each other via interfaces, the internal structures are omitted and the processes of π-calculus are used to describe the behaviors of interfaces to analyze the consistency of dynamic software architecture. To verify the structure characteristics (Yu et al., 2007), the π-calculus models are mapped into Petri nets to analyze the liveness of models.

5.1 The Compatibility of The Internal Implementations and Interfaces in Components

A component is composed of interfaces and internal implementations. Interfaces represent the providing or needing services of components. The internal implementations of components must be compatible with their interfaces. Therefore, components must execute the behaviors which interfaces represent. In this section, the name hiding (Canal et al., 2001) of π-calculus are used to analyze the compatibility.
Definition 4 (Name Hiding) Suppose a process P and a set of names $N \subseteq fn(P)$, P/N represents the process hides the names belong to N in P,

$$P/N = (\nu N)(P| \prod_{n \in N} Hide(n)) \tag{10}$$

where $Hide(n) = n(m).(Hide(n)|Hide(m)) + (\nu m)(\overline{n}\langle m\rangle.(Hide(n)|Hide(m)))$.
For each name $n \in N$, the process $Hide(n)$ hides it in P. $Hide(n)$ means that the input or output prefixes of n are provided to interact with the input or output prefixes of n in P. Therefore the behavior is predigested to the internal behavior τ so as to hide the name n.
The hide names are distinct with the restricted names. The restricted names cannot interact with the other processes, but they can be regarded as values to transmit to the other processes. The hide names means that the freedom names in processes are hide to transfer some specific behaviors to internal behaviors.
Definition 5 (Component Interface) Suppose a component $Comp$ and a process P, if

$$fn(P) \subseteq fn(Comp) \text{ and } P \approx Comp/(fn(Comp) - fn(p)) \tag{11}$$

then P is called an interface in $Comp$.
Definition 5 means that a interface is a subset of internal implementations in a component. Therefore the freedom names of the interface is a subset of the freedom names of the component, and the other names are hidden. Then the process is weak equivalent to P.
According to definition 5, the interfaces can be obtained by name hiding. However all processes satisfying definition 6 do not correctly represent the internal implementations of a component. The compatibility of the interfaces and implementations of a component is judged by the following conclusion.
Conclusion 1 Suppose a component $Comp$ and its set of interfaces $P = \{P_1, \ldots, P_n\}$ satisfy the following conditions.

1. $fn(P_i) \cap fn(P_j) = \varnothing \ \ \forall i \neq j$;
2. $Comp \Rightarrow 0$ iff $\forall i, P_i \Rightarrow 0$($\Rightarrow$represents $(\xrightarrow{\tau})^*$, which shows 0 or some internal evolvement sequence);
3. If $\exists \alpha(\alpha \neq \tau), Comp \xrightarrow{\alpha} Comp'$, then $\exists i, P_i', P_i \xrightarrow{\alpha} P_i'$ and the internal implementations of $Comp'$ are compatible with $P' = \{P_1, \ldots, P_i', \ldots, P_n\}$.

Then the internal implementations are compatible with the interfaces P.

5.2 Analyzing Consistency of Software Architecture

When new components join or components update, software architecture will evolve. Consequently, the consistency may be changed. The consistency means that all components in systems successfully interact with each other, which is important for dynamic software architecture (Goudarzi, 1998) (Canal et al., 2001). In SAFM, components interact with each other

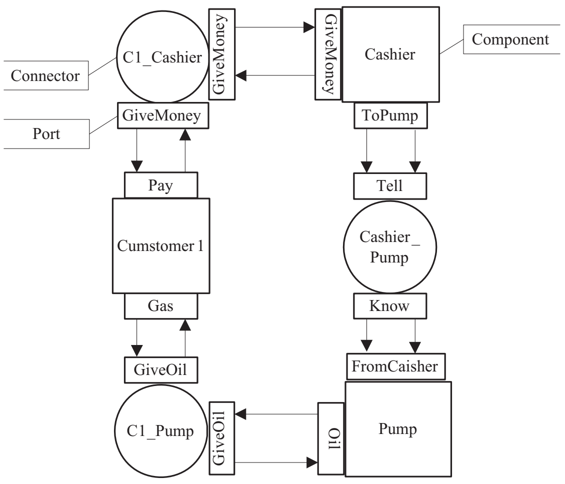

Fig. 3. software architecture model of gas station based on Wright

via interfaces. Therefore, the consistency can be judged from the interface level (Cimpan et al, 2005).

If the processes P and Q are consistent, P interacts with Q via a allelomorph name, such as $P = \overline{x}\langle y\rangle.P',\ Q = x(z).Q'$. Then P and Q are synchronous.

Definition 6(Consistency) Suppose the relation R in synchronous processes, for example PRQ. If all replacer operators $\sigma \notin fn(P) \cup fn(Q)$ satisfies $P\sigma RQ\sigma$, and

1. If $P \xrightarrow{\tau} P'$, then $P'RQ$;
2. If $\neg(P \sim 0) \wedge \neg(P \equiv 0),\ P \xrightarrow{x(z)} P' \wedge Q \xrightarrow{\overline{x}y} Q'$, then $P'\{y/z\}RQ'$;
3. If $\neg(P \sim 0) \wedge \neg(P \equiv 0),\ P \xrightarrow{x(n)} P' \wedge Q \xrightarrow{\overline{x}(n)} Q'$, then $P'RQ'$;

Then R is called half-consistency. If R and R^{-1} are half-consistency, then they are consistency, denoted as $\backsim$.

The consistency of processes means the processes can communicate with allelomorph names and execute successfully until their final states. If a process P' makes $P \Rightarrow P'$ and $P' \overset{\tau}{\not\rightarrow}$ or $P' \equiv 0$ ($P' \xrightarrow{\tau}$ means $\exists P'',\ P' \xrightarrow{\tau} P''$), then P can successfully execute until its final states. If $P \Rightarrow P',\ \neg(P' \xrightarrow{\tau})$ and $\neg(P' \equiv 0)$, then P is deadlock.

Conclusion 2(Consistency of Components) Suppose P and Q are interfaces of $Comp_1$ and $Comp_2$, and P and Q are consistent, $P \backsim Q$. Then $Comp_1$ and $Comp_2$ are consistent, namely, $Comp_1|Comp_2$ can successfully execute.

Conclusion 3(Consistency of System) If all components in a system are consistent, then the system can successfully execute.

6. Example: Modeling and Analyzing Gas-Station Problem

The gas station system consists of customers, cashiers and pumps (Tsai & Xu, 1999). If a customer arrives at the gas station, he pays for the gas, then the cashier informs the pump.

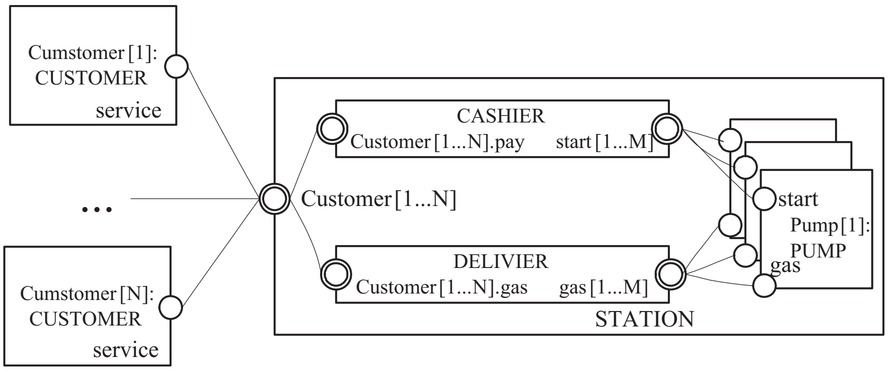

Fig. 4. software architecture model of gas station based on Darwin

Suppose that there exist a customer, a cashier, and a pump in gas station. The software architecture of the gas station using Wright is shown in Fig. 3 (Naumovich et al., 1997), where the arrows represent the connecting attachment. For this gas station, 3 connectors and 12 attachments are needed using Wright, and each connector, component, and port must be described by CSP processes. Therefore, the system model is complicated. The gas station system is modeled by Darwin shown in Fig. 4 (Magee et al., 1999). There no exist connectors in this system. Customers use the same port to pay for expenses and pump oil. Consequently collision may be occurred (Cuesta et al., 2005).
In this section, the gas station is modeled by SAFM shown in Fig. 5, which is abstracted as customer, cashier, pump and connector. The customer, cashier and pump are enrolled in the connector via the transition $RegInfo$. The connector sets up the pay_1, $pump_1$ and $info$ via $SevInfo$. Comparing to Wright and Darwin, the system model using SAFM is visual and easily understand, and void the collision in Darwin.
The Petri nets supporting tools– INA can be employed to analyze and verify the software architecture model of gas station. According to the INA analysis results, the model shown in Fig. 5 is bounded, the number of reachable states is 120, and it is live. Therefore, the gas station can successfully execute.
When a new customer arrives or the system provides new functions, the architecture evolves. Suppose that a new customer arrives, it firstly enroll in connector and the channel pay_2, $pump_2$ and $change$ are set up. Finally, according to the consistency of components, the consistency of the system needs to be analyzed. The interfaces of components in gas station are defined as follows.
$Customer_2$:

$$Pay(money) = \overline{pay_2}\langle money\rangle.(change(u).Pay(money) + paymore(p).Pay(p)) \tag{12}$$

$$PumpGas(x) = pump_2(x).PumpGas(x) \tag{13}$$

$Cashier$:

$$Charge(y) = pay_2(y).([y \geq expense]\overline{change}\langle dibs\rangle.Charge(y) + [y < expense] \overline{paymore}\langle expense - y\rangle.Charge(expense - y)) \tag{14}$$

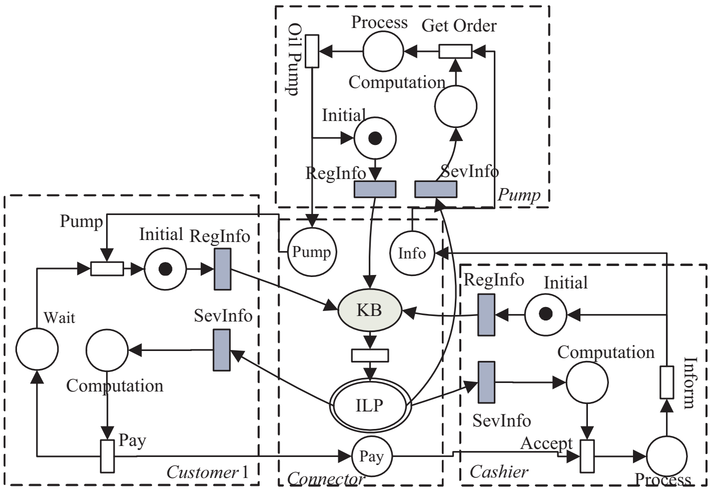

Fig. 5. software architecture model of gas station based on SAFM

$$Inform(id, msg) = \overline{info}\langle id, msg\rangle.Inform(id, msg) \tag{15}$$

Pump:

$$Getinfo(z, w) = info(z, w).Getinfo(z, w) \tag{16}$$

$$Pump(gas) = \overline{pump_2}\langle gas\rangle.Pump(gas) \tag{17}$$

The former interfaces of customer and cashier are

$$Pay'(money) = \overline{pay}\langle money\rangle.Pay'(money) \tag{18}$$

$$Charge'(y) = pay(y).Charge'(y) \tag{19}$$

The interfaces of customer and cashier provides new functions. According to rule 2, the cashier components is extended update. According to definition 6, $Pay \backsim Charge$, $Inform \backsim Getinfo$, $Inputgas \backsim Pump$, then components $Customer_2|Cashier|Pump$ can successfully execute. Consequently the gas station can run.

To analyze and verify the evolved model, it is translated into Petri nets by using the algorithm of π-calculus mapping Petri nets, which is shown in Fig. 6. The pump component is represented as an abstract place. The channels pay_2, $pump_2$, $change$ and $repay$ are mapped into places in the connector, and the interface transitions are added between the customer and cashier. INA is used to analyze the evolved model. According to the INA analysis results, the evolved model is deadlock, and there exists a dead reachable state S_{137}. By tracing the firing sequence of S_{137} and analyzing the system flow, an arc is added between T_{18} and P_{10}. Finally, the revised model is analyzed by INA. According to the INA analysis results, the revised model is bounded, the number of reachable states is 168, and it is live. Therefore, the system can successfully execute.

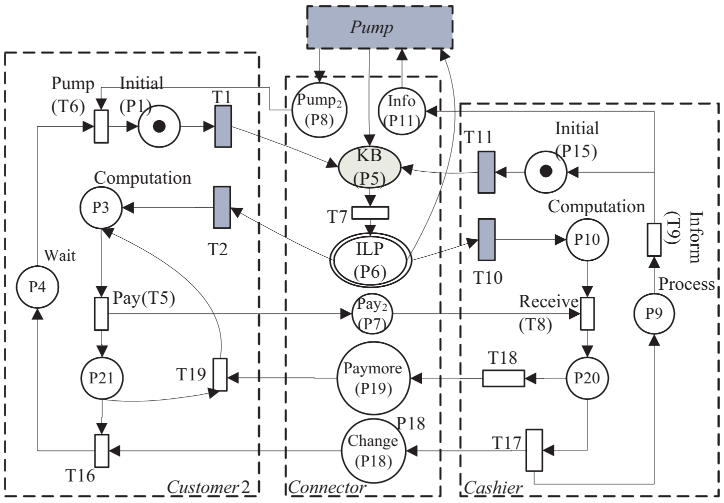

Fig. 6. the evolved software architecture model

According to the above model, the prototype system has been finished. SAFM also is applied to an engagement among Unmanned Ground Vehicles(UGV) (Yu, 2006).

7. Conclusion

Based on two complementary formalisms, namely Object-oriented Petri nets and π-calculus, software architecture formal model (SAFM) is proposed, which describes the components, connectors and configuration. OPN are employed to visualize the static architecture and depict the behavior of software systems, and π-calculus is used to describe software architecture evolution. SAFM stresses description of dynamic software architecture, and analyze the static and dynamic semantics, and depict the overall and individual characteristics of software architecture. SAFM can be applied to investigate software architecture from the micro-level and macro-level. From the micro-level, the researchers can pay more attention to the implementation details of each component; and from the society level, they can pay more attention to the overall design and interactions among components. An example of software architecture is used to illustrate modeling capability of SAFM.

8. References

Shaw, M.; Clements, P. (2006). The golden age of software architecture. *IEEE Software*, Vol. 23, No. 2: 31-39.

Oquendo F. (2006). π-ADL: an architecture description language based on the higher-order typed π-calculus for specifying dynamic and mobile software architectures. *ACM Software Engineering Notes*, Vol. 29, No. 4: 1-13.

Medvidovic, N.; Taylor, R. N. (2000). A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, Vol. 26, No. 1: 70-93.

He, X.; Yu, H.; Shi, T.; Ding, J.; et al.(2004). Formally analyzing software architectural specifications using SAM. *Journal of Systems and Software*, Vol. 71, No. 1-2: 11-29.

Shaw, M.; DeLine, R.; Klein, D. V.; et al. (1995). Abstractions for software architecture and tools to support them. *IEEE Transactions on Software Engineering*, Vol. 21, No. 4: 314-335.

Magee, J.; Delay, N.; Eisenbach, S.; Kramer, J. (1995). Specifying distributed software architectures, *Proceedings of the 5th European Software Engineering Conference*, Sitges.

Luckham, D. C.; Kenney, J. J.; Augustin, L. M.; et al. (1995). Specification and analysis of system architecture using Rapide. *IEEE Transactions on Software Engineering*, Vol. 21, No. 4: 336-355.

Allen, R.; Douence, R. & Garlan, D. (1998). Specifying and analyzing dynamic software architectures. *Lecture Notes in Computer Science*, Vol. 1382.

Luo H., Tang Z. & Zheng J. (2000). Visual architecture description language XYZ/ADL. *Journal of Software*, Vol. 11, No. 8: 1024-1029. (in Chinese with English abstract).

Yu, Z.; Cai, Y.; & Xu, H. (2007). On Petri nets semantics for π-calculus. *Control and Decision*, Vol. 22, No. 8: 864-868. (in Chinese with English abstract).

Miyamoto, T. & Kumagai S. (2005). A survey of Object-Oriented Petri nets and analysis methods. *IEICE Transaction on Fundamentals*, Vol. E88-A, No. 11: 2964-2971.

Yu, Z. (2006). *Approaches to formal modeling methodology for multi-agent systems* [Ph.D. Thesis]. Xi'an: Xi'an Jiaotong University.

Milner, R.; Parrow, J. & Walker D. (1992). A calculus of mobile processes. *Journal of Information and Computation*, Vol. 100, No. 1: 1-77.

Milner, R. (1993). *The polyadic π-calculus: a tutorial*, Springer-verlag.

Jiang, C. (2003). *Behvior theory and applications of Petri net*, Beijing: Higher Education Press.

Clarke, E. & Wing J. (1996).Formal methods: state of the art and future. *ACM Computing Surveys*, Vol. 28, No. 4: 626-643.

Best, E.; Devillers, R. & Koutny, M. (2001). *Petri Net Algebra*, Spring-Verlag.

Canal, C.; Pimentel, E. & Troya J. M. (2001). Compatibility and inheritance in software architectures. *Science of Computer Programming*, Vol. 41, No. 9: 105-138.

Goudarzi, K. (1998). *Consistency preserving dynamic reconfiguration of distributed systems* [Ph. D. thesis]. Imperial College London.

Cimpan, S.; Leymonerie, F. & Flavio Oquendo, F. (2005). Handling dynamic behaviour in software architectures. *Lecture Notes in Computer Science* , Vol. 3527: 77-93.

Tsai, J. J. P. & Xu, K. (1999). An empirical evaluation of deadlock detection in software architecture specifications. *Annals of Software Engineering*, Vol. 7, No. (1-4): 95-126.

Ma, X.; Yu, P.; Tao, X.; Lv, J. (2005). A service-oriented dynamic coordination architecture and its supporting system. *Chinese Journal of Computer*, Vol. 28, No. 4: 467-477.

Cuesta, C. E.; de la Fuente, P., Barrio-Solorzano, M.; Beato, M. E. (2005). An "abstract process" approach to algebraic dynamic architecture description. *Journal of Logic and Algebraic Programming*, Vol. 63, No. 2: 177-214.

Canal, C.; Pimentel, E. & Troya, J. M. (1999). *Software architectures*. Kluwer Academic Publishers.

Roch, S. & Starke P. H. (2009). *INA: integrated net analyzer, Version 2.2*, www2.informatik.huberlin.de/˜ starke/ina.html.

Naumovich, G.; Avrunin, G. S.; Clarke, L. A.; Osterweil, L. J. (1997). Applying static analysis to software architectures. *Lecture Notes in Computer Science*, Vol. 1301, 77-93.

Magee, J.; Kramer, J. & Giannakopoulou D. (1999). *Behaviour analysis of software architectures.* Kluwer Academic Publishers: Software Architecture.