Automated Adaptations to Dynamic Software Architectures
by Using Autonomous Agents

Wenpin Jiao*, Hong Mei
Institute of Software, School of Electronics Engineering and Computer Science
Peking University, Beijing 100871, China
jwp@cs.pku.edu.cn, meih@pku.edu.cn

Abstract: Software architectures should always reflect the architectural changes occurring in software systems. In this paper, dynamic software architectures are studied from a new perspective. Software architectures are considered as goals or objectives that software systems are pursuing instead of constraints over systems. The architectural goals are decomposed into two categories of sub-goals, i.e., structural goals and interaction behavioral goals. For defining the architectural goals of software systems, the notation of architectural reference model is introduced. The architectural reference model is composed of components and connectors involved in software architectures and constraints over software architectures. For reasoning about the result of changes rigorously and automatically, software architectures and changes to software architectures are described in a uniform formalism and the results of changes can directly be derived from the compositions of software architectures and changes. For implementing the adaptations of software systems, an autonomous agent based approach is adopted, in which autonomous agents are used to plan the achievement of architectural goals and cascade reactions caused by changes via using architectural style-specific knowledge and behavior rules defined for agents.

Keywords: Dynamic Software Architectures, Adaptation, Goal-Driven, Autonomous Agents, pi-Calculus

1. Introduction

Today, more and more software systems are required to be able to adapt themselves at run time to handle changes resulted from dynamic user requirements or environments. As the blueprints of software systems, software architectures

* Corresponding author: Institute of Software, School of Electronics Engineering and Computer Science, Peking University, Beijing 100871, P. R. China. Tel.: +86-10-62757670; fax: +86-10-62751792. E-mail address: jwp@cs.pku.edu.cn.
should eventually reflect those changes through architectural operations such as adding/removing components/connectors or reconfiguring. Study on adaptations to dynamic software architectures at runtime will benefit greatly on the improvement of the capability of adaptations of software systems and has been paid more and more attention in the software engineering community.

Even though the requirement for adaptation is becoming more widespread, current researches on adaptations to dynamic software architectures are still mostly ad hoc and the adaptations are typically provided in a per-system manner and embedded in the application code. Consequently, it is difficult to extract knowledge about adaptations from application systems and to analyze further to ascertain whether the adaptations will result in proper-running systems (Schmerl and Garlan, 2002).

To deal with the adaptations to dynamic software architectures rigorously, approaches need abilities such as:

- To describe software architectures and express changes in the same way so that how changes take effects on the evolutions of software architectures could be formally reasoned about.
- By composing the changes and the original software architectures together, to produce the new software architectures automatically.
- To analyze and verify whether the new produced software architectures conform to the desired or expected software architectures.
- And at last, to make changes on the software systems effective and generate the desired software architectures automatically.

Generally, architecture description languages (ADLs) provide a means of specifying software architectures formally. However, most of existing ADLs (e.g., ABC/ADL (Mei et al., 2002), ACME (Garlan et al., 2000), Darwin (Magee and Kramer, 1996), Rapide (Luckham et al., 1995), Wright (Allen and Garlan, 1997), UniCon (Shaw et al., 1995)) are mainly used to define those static properties of software architectures though there are some exceptions. For instance, C2 SADL (Medvidovic et al., 1997) and CommUnity (Wermelinger and Fiadeiro, 2000) define a set of architectural operations for manipulating software architectures. However, no such approach can directly reason about the impacts that changes may put on software architectures. Although Dashofy et al (2002) and Van der Westhuizen and Van der Hoek (2002) propose an approach to understand and analyze the results of architectural changes, the approach lacks of a formal foundation.

Meanwhile, there has been much research on adaptability (e.g., Decker and Sycara, 1997; Brazier and Wijngaards, 2001; Fatima and Uma, 1998; Jennings, 1996; Tambe, 1997). To embody adaptation in a software system, the techniques they used are usually based on machine learning or decision-making (Barber et al., 2000), but many of them are focused on centralized processes to formulate models or strategies. Meanwhile, there has also been much practice on developing adaptive software systems (Tambe et al., 2000). However, most of those researches focus on the adaptation
of individual entities.

Especially, in the study on the adaptations of software systems, software architectures are generally treated as structural constraints over the evolutions of systems, i.e., the evolutions of systems should maintain software architectures (or architectural styles) to be unchanged (Cheng et al., 2002; Garlan and Schmerl, 2002). In some literature, the adaptations of software systems are studied via exploring the impacts of changes on software systems (de Paula et al., 1999; Medvidovic, 1996; Rana et al., 2003), in which operations coping with changes are defined to manipulate software systems and then the results of the manipulations are analyzed to reason about whether the evolved systems still satisfy the constraints specified for software architectures.

To some extends, architectures of software systems can be considered as the design goals of software systems besides specifying the structures or configurations of systems. While software systems are being developed, they are often required being implemented in some specific control or interaction structures. For instance, a system may be asked to implement in a client/service style or with a workflow control structure. That is to say, while software systems are being realized, some architectural requirements may have to be satisfied, i.e., the systems should be implemented in some specified software architectures, besides those traditional functional or non-functional requirements.

In this paper, by treating software architectures as goals that software systems pursue while they are being developed and executed, this paper puts forward a strategy based on autonomic computing technology to exploit self-adaptations to dynamic software architectures and implement self-adaptations by using autonomous agents.

From the perspectives of automatic computing (Kephart and Chess, 2003), a self-adaptive software system must be aware of the boundary of itself. Only when a self-adaptive system knows both itself and its environment well can it distinguish which influences are resulted from itself or from its environment and then take proper reactions to those influences to adapt its behaviors or configuration. Autonomy, pro-activity, and goal-directed interactivity with their environment are distinguishing characteristics of software agents. This is the main reason that agents are adopted as the means of coping with the self-adaptations of software systems to dynamic software architectures since agents are entities knowing both themselves and their environment.

Generally considering, agents are autonomous computer (software) systems situated in environments (Wooldridge and Jennings, 1995). As autonomous entities, agents firstly know clearly what they want to do or what their goals are and secondly can perceive changes taking place in their environments. So the implementation of an autonomous agent often contains a model for defining the internal mental states of an agent as well as a model for describing the external world (or environment) (e.g., Muller, 1997). The former model implies the purpose of an agent’s existence and the latter indicates how the external environment changes and what changes an agent may be sensitive to.

By implementing self-adaptive systems as autonomous agents, the ability of environmental awareness of agents can be used to obtain the changes appearing in the environments and reason about the impacts of the changes.
Meanwhile, by modeling self-adaptive systems as internal models of autonomous agents, both implementing specified software architectures and self-adapting to dynamic software architectures can naturally be considered as goals of autonomous agents.

Thus, for an agent-based software system, the self-adaptation of the system will be exhibited through a sequence of activities of agents, such as sensing changes of the environment, obtaining goals for adaptations, and planning evolutions in response to the changes.

Then, the evolutions of the software system will be reflected via those changes to the model of the system itself. And furthermore, the self-adaptation of an agent-based software system will be able to be represented via autonomic manipulations on the model of the system.

This paper will probe into the issues such as how to reason about the impacts that changes put on to software architectures, how to verify that the results of changes achieve the desired goals, and how to implement the automated update to software architectures. The main contributions of this paper are on four aspects.

1). Software architectures are inspected and investigated from a new perspective, i.e., software architectures are considered as goals or objectives that software systems are pursuing.

In this work, software architectures are viewed as a kind of non-functional goals that software systems are pursuing, i.e., the design goals, and further architectural goals are specified as compounds of structural goals and interaction behavioral goals. The structural goals specify how components should be connected with each other and the interaction behavioral goals stipulate how components should take actions to interact with one another, or in other words, what interaction protocols components should abide by.

2). Software architectures and changes are expressed in a uniform formalism (e.g., the pi-calculus) and then based on the capability provided by the pi-calculus, the impacts of changes on software architectures can be reasoned about and verified automatically.

For reasoning about the impacts automatically, it is necessary to define both software architectures and changes in the same formal framework. So a subset of the pi-calculus (Milner et al., 1992) is selected as the formalism to describe software architectures and changes. The pi-calculus has been proved to be a very expressive notation for describing the behaviors of software components in applications with changing topologies. More importantly, changes are often expressed in the forms of sequences of actions and thus can be easily defined as processes in the pi-calculus.

3). Based on the bi-simulation theory of the pi-calculus processes, whether architectural changes can lead to valid software architectures and the results of architectural changes will satisfy the desired requirement or achieve the expected architectural goals can be proved and verified.

4). Automated updates on software architectures are implemented by adopting an autonomous agent-based approach.
First, to realize specific software architectures are taken as goals of autonomous agents and those autonomous agents are responsible for making plans to achieve the goals, *i.e.*, to determine what components are needed and how components are connected and interact.

Secondly, to adapt to the dynamic changes of software architectures, autonomous agents will sense and monitor the changes of software architectures.

Generally, changes may cause cascade reactions. For instance, if the change of removing a connector will result in isolating a component from a software system, the affected component should be removed from the system, too, or be provided a new connection service via another connector. So thirdly, autonomous agents will plan on how to make changes to software architectures and how to handle cascade reactions to ensure that changed software architectures are still in valid states and the architectural goals can be achieved.

In the following context, Section 2 simply introduces the formalism that will be used to formalize software architecture and its elements, *i.e.*, the pi-calculus. Section 3 formally describes goal-driven software architectures. For defining the goals of software architectures, a notation of architectural reference model is introduced first. Section 4 and Section 5 describe changes and the impacts caused by changes. Based on those formal foundation studied in previous sections, Section 6 describes the autonomous agent-based implementation for adaptations to dynamic software architectures. In Section 6 and Section 7, two examples are given to show how autonomous agents work to adapt to the changes of software architectures. In Section 8 and Section 9, some related work is compared and some conclusion remarks are discussed.

2. The pi-calculus

The pi-calculus (Milner *et al.*, 1992) is a model of concurrent computation based upon the notion of naming, and it is a way of describing and analyzing systems consisting of agents which interact among each other, and whose configuration or neighborhood is continually changing. Processes that have changing structures can naturally be expressed by using the pi-calculus.

In the pi-calculus, there are only two kinds of entities: processes and channels. Processes are active components of a system and a channel is the link implicitly defined between two ports (an import port and an export port with complementary names, *e.g.*, $\alpha$ and $\overline{\alpha}$). Processes communicate with each other through ports connected via channels. In the pi-calculus, all of data, ports and variables are names; and communications of sending/receiving messages (or inputting/outputting names) between processes take place through import/export ports. The processes in the pi-calculus have the following forms.
\[
\begin{align*}
    P := 0 & \quad \pi \cdot P \quad P | Q \quad !P \quad (\nu x)P \quad [x = y]P \\
    \pi := x(y) | \overline{xy} | \tau
\end{align*}
\]

\(x(y)\) and \(\overline{x(y)}\) respectively represent input and output actions where \(x\) is the link (or channel) along which the actions are performed and \(y\) is a name (either link or data) sent or received along \(x\). \(\tau\) denotes an internal (unobservable) action while the special process 0 represents inaction. \(P|Q\) represents the parallel composition of two processes of \(P\) and \(Q\). When \(P = x(y) \cdot P'\) and \(Q = \overline{x} \cdot Q'\), the parallel composition of these processes may synchronize on \(x\) and then yield the derivative \(P'[z/y]|Q'\), where \(P'[z/y]\) means all free occurrences of \(y\) in \(P\) will be substituted by \(z\). The above parallel computation can be represented via the reduction rule as

\[
\frac{x(y) \cdot P | \overline{x} \cdot Q \xrightarrow{\nu z \in \text{Vg}} P'[z/y] | Q'}{!P}
\]

\(P\) represents any number of copies of \(P\), i.e., \(!P = P \cdot P \cdot \ldots\). \((\nu x)P\) represents introducing a new channel \(x\) with scope \(P\), where \(v\) is the constriction operator. In \((\nu x)P\), if \(x\) is the name of an action, all input/output actions named with \(x\) and \(\overline{x}\) will be unobservable outside of \(P\); and if \(x\) is the name of a variable, the variable is private to \(P\). \([x = y]P\) represents that process \(P\) will proceed if \(x\) and \(y\) are the same channel, where \([x = y]\) is the matching operator.

### 2.1. The Transitional Semantics

The computations (or evolutions) of the pi-calculus processes are usually expressed via reduction rules. In the following rules, \(\xrightarrow{\nu z} \) represents a reduction by which a process reduces to another process after an action (or a parallel composition of actions) such as \(x(y)\), \(\overline{xy}\), and \(\tau\).

1. **Communication rule**

   In the pi-calculus, computation is expressed by the following communication rule.

   \[
   \text{COMM:} \quad (\cdots + x(y) \cdot P) | (\cdots + \overline{x}(z) \cdot Q) \xrightarrow{\nu z \in \text{Vg}} P'[z/y] | Q\]

   This means sending name \(z\) along link \(x\) reduces the left-hand side to \(P|Q\) with all free occurrences of \(y\) in \(P\) replaced by \(z\).

2. **Parallel rule**

   Action between two parallel processes can be expressed by the following parallel rule.

   \[
   \text{PAR:} \quad P \rightarrow P' \\
   \frac{P | Q \rightarrow P' | Q'}{P | Q \rightarrow P' | Q'}
   \]

   It means if there is no communication between the two processes \(P\) and \(Q\), their actions are interleaving.

3. **Restriction Rule**

   \[
   \text{RES:} \quad P \rightarrow P' \\
   \frac{(\nu x)P \rightarrow (\nu x)P'}{(\nu x)P \rightarrow (\nu x)P'}
   \]
It means restriction by a name, which does not occur freely in a process, does not affect the behavior of the process.

4. Structural Congruence Rule

\[
\text{STRUCT: } Q = P, P \rightarrow P', P' = Q' \\
\frac{Q \rightarrow Q'}{}
\]

(5)

It means if there are two structural congruence processes, they will act in the same way.

2.2. Simulation and Equivalence

In the pi-calculus, for studying whether a process will execute in the expected way, simulation relations are defined to indicate that a process is similar to (or simulates the behavior of) another reference process.

Informally, a process is said to be (strongly) similar to another process if the former can always exhibit the exactly same sequence of actions as the latter. If the internal communications or unobservable actions exposed by the two processes are not taken into consideration, the former can be considered to be weakly similar to the latter.

Then binary relations for describing the similarities of processes can be formally defined as follows.

**Definition 1** (Sewell, 2000). A binary relation \( S \) on processes is a strong simulation if it satisfies the following requirements: \( P \mathrel{S} Q \) and \( P \xrightarrow{\tau} P' \), where \( \pi = x(y) \), or \( \pi(y) \) and \( y \) is a fresh name relative to both \( P \) and \( Q \) (A name is fresh relative to a process means that the name is different from any free names occurring in the process.), implies that

i). If \( \pi = x(y) \), then

\[ \exists \overline{Q'}: Q \xrightarrow{\pi(y)} Q' \land P \!\downarrow \! y \vdash u: P'[u / y] \mathrel{S} Q'[u / y] ; \]

ii). Otherwise, \( \exists \overline{Q'}: Q \xrightarrow{\pi} Q' \land P' \mathrel{S} Q' \).

\( S \) is a strong bi-simulation if both \( S \) and its inverse are simulations. Two processes \( P \) and \( Q \) are bi-similar if and only if there exists a bi-simulation \( S \) such that \( P \mathrel{S} Q \).

In the pi-calculus, a silent action \( \tau \) means that the action is unobservable from the outside. Since \( x \mid \overline{x} \xrightarrow{\tau} \tau \) by referring to the reduction rule mentioned about, internal communications within processes are unobservable, either. If only those observable actions executed by processes are taken into consideration, the weak bi-simulation relation can be defined as follows.

**Definition 2**. Let \( \pi^* \) denote \( \pi \) begun or followed with a sequence of \( \tau \). Then a binary relation \( \mathcal{W} \) on processes is a weak simulation if it satisfies those requirements described in the definition of the strong simulation relation after replacing \( \pi \) by \( \pi^* \). Similarly, weak bi-simulation and weak bi-simulation can be defined for \( P \) and \( Q \).
2.3. Negative and Simplified Process

While components and connectors are defined in the pi-calculus, it should be ensured that components could be properly connected to connectors. The notation, negative process, is introduced for achieving this (Jiao and Mei, 2003).

**Definition 3.** The negative of process \( P \), \( \neg P \), is recursively defined as follows by referring to the definition of the pi-calculus processes.

i). The negatives of atomic actions \( x(y), \overline{x}(y), \) and \( \tau \) are \( \overline{x(u), x(u),} \) and \( \tau \) for any \( u \), respectively, *i.e.*, the negative of \( \pi \) is \( \neg\pi \);

ii). If \( P = \pi \cdot Q \), then \( \neg P = \neg\pi \cdot \neg Q \);

iii). If \( P = Q|R \), then \( \neg P = \neg Q|\neg R \);

iv). If \( P = !Q \), then \( \neg P = !\neg Q \);

v). If \( P = (\forall x)Q \), then \( \neg P = (\forall x)\neg Q \);

vi). If \( P = [x = y]Q \), then \( \neg P = [x = y]\neg Q \).

For the negative relation, \( \neg\neg P = P \). And \( P \) is also called as its negative’s positive process.

Intuitively, the negative of a process and the process itself can communicate and reduce eventually and exactly into the inert process \( 0 \). The following outcome about the relation on a process and its negative can be obtained.

**Corollary 1.** If \( P \) will terminate eventually (e.g., the number of its replica is finite), then \( P \mid \neg P \) will reduce into \( 0 \) after finite steps of communications.

**Definition 4. Restricted Form.** Suppose \( x_1, x_2, \ldots, x_n \) are unbound names (*i.e.*, free names) occurring in \( P \). Then \( P_{\text{restricted}} = (\forall x_1, x_2, \ldots, x_n)P \) is the restricted form of \( P \).

Intuitively, the restricted form of a process makes all actions occurring in the process unobservable.

**Definition 5. Simplified Form.** Suppose \( P \) is recursively defined via a function \( f \) as \( P = f(x_1, x_2, \ldots, x_n, P) \). Then \( P_{\text{simplified}} = f(x_1, x_2, \ldots, x_n, \tau) \) is the simplified form of \( P \). Furthermore, if \( P = f(Q) \), then \( P_{\text{simplified}} = f(Q_{\text{simplified}}) \).

Informally, the simplification of a process will lead a non-terminating process to be terminable by removing the recursions.

**Corollary 2.** \( P_{\text{simplified}} \mid (\neg P)_{\text{simplified}} \) will reduce into \( 0 \) after finite steps of communications.

2.4. Why the pi-Calculus

Since this research work is focusing on the adaptations of software systems to dynamic software architectures, the selected formalism should be competent with describing and coping with the dynamic evolutions of software systems and their dynamic architectures. Coincidently, as mentioned above, the pi-calculus is very suitable to describe software systems with changing configurations and also powerful to express the evolutions of software systems. So the pi-
calculus is a very natural selection used for formalizing dynamic software architectures.

Secondly, the connections between components and connectors are dynamic. That is, in different situation and at different time, a component may connect to other components via different connector. A good formalism had better make the process of dynamically connecting components and connectors transparent. Not like most of ADLs, for instance, Wright (Allen and Garlan, 1997), in which the attachments of components and connectors should be explicitly defined, the pi-calculus provides means of transporting links between processes, which can be used for establishing dynamic connections between components and connectors. In the following descriptions, this feature of the pi-calculus will be used to implement the transmission of import/export ports of components to connectors for building dynamic connections.

Thirdly, evolutions are always emerging from changes. For software architectures, changes are actions to manipulate or operate the software architectures and then the evolutions of dynamic software architectures are embodied in the effects of the actions. Therefore, reasoning about and analyzing the evolutions of dynamic software architectures is in fact to reason about actions and their effects. However, traditional formal methods (such as first-order logics and temporal logics) can only describe constraints over actions instead of describing actions directly, say nothing of reasoning about actions. Contrarily, the pi-calculus cannot only describe actions but also reason about actions. Thus, the pi-calculus provides us a uniform approach to specify dynamic software architectures and reason about the evolutions of dynamic software architectures.

Fourthly, reasoning about the evolutions of dynamic software architectures has two aspects, one is about how software architectures evolve and the other is about whether the evolutions are valid, i.e., whether the evolutions lead to desired or expected software architectures. Conventionally, the second aspect of reasoning is done by using methods based on logics (e.g., in Dashofy et al., 2002), which is usually too complicated to be automated. Differently, in the pi-calculus, the bi-similarity theory is built for reasoning about whether the implementation of a system conforms to the specification of the system and the reasoning can be easily handled via observing the behaviors exposed by the systems. In this paper, just a subset of the pi-calculus to describe and reason about dynamic software architectures is used, i.e., omitting the operator of non-deterministic choice among processes (refer to Milner et al., 1992 for a complete definition for the pi-calculus processes). And thus the reasoning based on bi-similarity will turn to be simpler and possible to be automated.
3. Goal-Driven Software Architectures

For defining architectural goals of software systems in the pi-calculus, a notation of architectural reference model for software architectures is defined first. Thus, the architectural goals can be defined in the pi-calculus via the simulation relations between software systems and their architectural reference models, in which the architectural goals are achieved if the architectures of software systems can simulate the reference models.

3.1. Software Architectural Reference Model

The kernels of software architectures are components, connectors, and connections among them, and software architectures can be reflected from appropriate connections or configurations of components and connectors. The simplest form of architectural reference models contains only one connector and a set of components that will connect together through the connector. And real software architectures can be treated as instances of the composition of a finite set of architectural reference models.

To define architectural reference models, here adopts a strategy different from those in the literature. Most of ADLs define software architectures into three parts, i.e., components, connectors, and configurations to attach components and connectors together. However, the paradigm used for describing configurations is often different from that for defining components and connectors. For instance, components and connectors are defined via using languages with special operational semantics (e.g., the Z notation, the pi-calculus, and the CSP.), whereas configurations are defined via mappings between port names, e.g., Wright (Allen and Garlan, 1997) and Darwin Darwin (Magee and Kramer, 1996). This will cause the description of a software architecture containing two or more kinds of different semantic forms. To avoid this situation, in the architectural reference models defined in the pi-calculus, configuration information is implicitly expressed within the definitions for components and connectors.

For making the description clearer, an example system is used to illustrate the approach. For example, there is a fault-tolerant client/server system shown as follows (This example refers to the case study in Allen et al., 1998). In such system, to prevent the system from going down because of the primary server’s failures, a backup (or secondary) server is always engaged in and ready for taking the place of the primary server to provide services once the primary server dies due to its failures. Fig. 1 shows the dynamics of the system’s configuration.

---

1 In some cases, the reference model of software architectures can also be considered as the architectural pattern of software. However, like (Luckham, 1995), the model in the approach described in this paper is the reference that the realizations of dynamic software architectures should be compatible with.
In this system, the architectural reference model includes two components, a client and a server, and a connector providing connection service for the client and the server.

### 3.1.1. Atomic Component

An atomic component is usually specified as a compound of its interface description and its behavior specification, in which the interface defines the ports used for connecting the component with a connector whilst the behavior specification specifies how the component uses the ports to interact with other components via the connector.

Conceptually, ports in the interface can be regarded as virtual arguments used in the behavior specification and will be bound to concrete ports of a connector after the component is connected to the connector. Though the behavior specification depends on the binding of ports in the interface, a component is still able to be defined as the parallel composition of its interface and its implementation.

Although a component may connect with more than one other component through connectors, any component can be considered as a composition of several atomic components that connect to a single connector.

**Definition 6.** An atomic component is supposed to connect to a single connector. Suppose that \{i_1, i_2, ..., i_n, o_1, o_2, ..., o_m\} is the set of ports defined in the interface of an atomic component \(P\), in which \(i_k\) is an input port and \(o_j\) is an output port. Then

\[
P = (\forall i_1, \cdots, i_n, o_1, \cdots, o_m) P_{\text{Interface}} \mid P_{\text{Spec}}(i_1, \cdots, i_n, o_1, \cdots, o_m)
\]

\[
P_{\text{Interface}} = \overline{p}(i_1, \cdots, i_n, o_1, \cdots, o_m)
\]

Where, \(P_{\text{Interface}}\) is the interface specification, in which \(\overline{p}(i_1, \cdots, i_n, o_1, \cdots, o_m)\) will transmit all of the names of ports to a connector for establishing a connection between the component and the connector. \(P_{\text{Spec}}\) specifies the behaviors of the component, which defines protocols that the component uses to interact with other components.

Since those ports of the component are defined as private names, the behavior specification can be executed only after the ports have been bound to a connector through the interface.
3.1.2. Atomic Connector

A connector is always defined to provide connection services for components. Concretely speaking, a connector should first know the interface specifications of components in order to connect with components properly. In addition, a connector may act as a media (or an adaptor) to provide seamless connections for components since components are often specified independently and there may be mismatches between their behaviors (e.g., mismatching interaction protocols (Yellin and Strom, 1997), or mismatching architectural styles (Jiao and Mei, 2003)).

For ensuring that components could properly connect together through connectors, the notation of negative process is used in the definition of connectors.

**Definition 7.** Suppose that connector \( C \) will provide connections for components \( P_1, P_2, \ldots, P_n \). Then \( C \) can be defined as the following pi-calculus process.

\[
C = \neg P_1_{\text{Interface}} \parallel \neg P_2_{\text{Interface}} \parallel \cdots \parallel \neg P_n_{\text{Interface}} \parallel C_{\text{Protocol}}
\]

(7)

Where, \( C_{\text{Protocol}} \) defines the protocols for the interactions among \( P_1, P_2, \ldots, P_n \).

In the definition, unlike components, it does not limit the number of components that a connector can connect. In some cases, for instance, in a C2-style based software architecture, a connector may provide connections for more than two components.

The relationship between components and connectors can be shown in Fig. 2 as follows.

![Fig. 2. Relationship between components and connectors](image)

As shown in Fig. 2, a connector comprises a collection of negative processes of components' interfaces so that, according to Corollary 2, the connector can properly connect with components, *i.e.*, can provide appropriate connection services for components.

**Definition 8.** *Well-established connection.* Suppose that connector \( C \) provides connections for components \( P_1, P_2, \ldots, P_n \). Then it says that \( P_1, P_2, \ldots, P_n \) can properly connect together through \( C \), *i.e.*, the connections among them are well-established, if \( (P_1 \parallel P_2 \parallel \cdots \parallel P_n \parallel C)_{\text{simplified}} \) can reduce into inert process \( 0 \), *i.e.*, \( (P_1 \parallel P_2 \parallel \cdots \parallel P_n \parallel C)_{\text{simplified}} \rightarrow^* 0 \).

3.1.3. Atomic Architectural Reference Model

An atomic reference model of software architectures can be defined as the composition of a connector and a group.
of components connecting to the connector.

**Definition 9.** Suppose there are a group of components \( P_1, P_2, \ldots, P_n \) and a connector \( C \) providing connections among these components. The architectural reference model, \( RM \), is defined as a composition process provided that \( P_1, P_2, \ldots, P_n \) can setup well-established connections through \( C \), i.e., \( RM = P_1 | P_2 | \cdots | P_n | C \). \( \Box \)

For example, in the fault-tolerant client/server system, a client and a server can be defined as follows, respectively.

\[
\text{Client} = (v \ c\_\text{reply}, c\_\text{request}) \text{client}(c\_\text{reply}, c\_\text{request}) | \text{Client}_{\text{spec}}(c\_\text{request}, c\_\text{reply})
\]

\[
\text{Client}_{\text{spec}} = (vx)IC \cdot c\_\text{request}(req) \cdot c\_\text{reply}(x) \cdot \text{Client}_{\text{spec}}
\]

Where, \( c\_\text{request}(x) \) sends out a request to the server and \( c\_\text{reply}(y) \) waits for a response from the server. \( IC \) represents the internal computation.

\[
\text{Server} = (v s\_\text{request}, s\_\text{reply}) \text{server}(s\_\text{request}, s\_\text{reply}) | \text{Server}_{\text{spec}}
\]

\[
\text{Server}_{\text{spec}} = (vy)s\_\text{request}(y) \cdot IC \cdot s\_\text{reply}(\text{respond}) \cdot \text{Server}_{\text{spec}}
\]

The connector between the client and the server can be defined as follows.

\[
\text{CS} = \neg \text{Client}_{\text{interface}} | \neg \text{Server}_{\text{interface}} | \text{CS}_{\text{protocol}}
\]

\[
= \text{client}(ci, co) | \text{server}(si, so) | \text{CS}_{\text{protocol}}
\]

\[
\text{CS}_{\text{protocol}} = (vxy)co(x) \cdot si(x) \cdot so(y) \cdot ci(y) \cdot \text{CS}_{\text{protocol}}
\]

When the client and the server are connected together through the connector, they will evolve as follows, in which the substitutions of ports (e.g., \( c\_\text{request}/co \)) also indicate that ports of components are attached with corresponding ports of the connector.

\[
\text{Client} | \text{Server} | \text{CS} \rightarrow^* \text{Client}_{\text{spec}} | \text{Server}_{\text{spec}} | \text{CS}_{\text{protocol}}(c\_\text{request}/co, c\_\text{reply}/ci, s\_\text{request}/si, s\_\text{reply}/so),
\]

and then \( (\text{Client} | \text{Server} | \text{CS})_{\text{Simplified}} \rightarrow^* 0 \).

By referring to the definition for well-established connection, some conclusions can be drawn, i.e., the client and the server can setup a well-established connection via the connector \( CS \) and furthermore \( \text{Client} | \text{Server} | \text{CS} \) defines a valid atomic architectural reference model for client/server style-based software architectures.

### 3.1.4. Composition of Reference Models

Real software architectures are often more complex than those defined by the atomic architectural reference models. Nevertheless, no matter how complex software architectures are, they can be viewed as instances of the compositions of atomic architectural reference models.

In the pi-calculus, processes can be composed only via parallel composition operations. If reference models are considered to be undividable, the composition of reference models \( RM_1 \) and \( RM_2 \) may only be with the form of \( RM_1 | RM_2 \). However, this kind of composition is too simple to reflect some special topological relationships between
components and connectors. So it is preferable to view architectural reference models as something dividable and with internal structures. Thus the compositions of architectural reference models can be divided into three categories. In compositions, new components or connectors may be formed and then treated as a whole.

1). Component-based Compositions. In this category, components from different architectural reference models may compose into a new component. After compositions, the result reference models may contain complex components that can connect to multiple connectors.

For example, suppose that \( RM_1 = P_1 \mid P_2 \mid C_1 \), \( RM_2 = P_3 \mid P_4 \mid C_2 \), and \( P_1 \) and \( P_3 \) will compose into a new component, called \( P \). Then the composition of \( RM_1 \) and \( RM_2 \) will look like \( RM' = P \mid P_2 \mid P_4 \mid C_1 \mid C_2 \), in which the new formed component \( P \) will connect to both \( C_1 \) and \( C_2 \) (see Fig. 3).

2). Connector-based Compositions. While reference models being composed, connectors from different architectural reference models may compose into a new connector.

For example, suppose that \( RM_1 = P_1 \mid P_2 \mid C_1 \), \( RM_2 = P_3 \mid P_4 \mid C_2 \), and \( C_1 \) and \( C_3 \) will compose into a new connector, called \( C \). Then the composition of \( RM_1 \) and \( RM_2 \) will be like \( RM' = P_1 \mid P_2 \mid P_3 \mid P_4 \mid C \), in which \( C \) will provide connections for all components (see Fig. 4).

3). Concatenation, i.e., architectural reference models are bound together simply through the parallel composition operation of the pi-calculus.

For example, suppose that \( RM_1 = P_1 \mid P_2 \mid C_1 \), and \( RM_2 = P_3 \mid P_4 \mid C_2 \). Then the composition of \( RM_1 \) and \( RM_2 \) will be like \( RM = P_1 \mid P_2 \mid P_3 \mid P_4 \mid C_1 \mid C_2 \).

Definition 10. Complete (or self-contained) Architectural Reference Model. An architectural reference model, \( RM = P_1 \mid \cdots \mid P_n \mid C_1 \mid \cdots \mid C_n \), is said to be complete if it meets the following requirements:

1). \( \forall i \in \{1, \ldots, n\} \exists j \in \{1, \ldots, n\} \quad C_j = \neg P_{\text{interface}} \mid C_j \), i.e., any components can connect to other components through connectors;

2). \( \forall i,j \in \{1, \ldots, n\} \quad C_i = \neg P_{\text{interface}} \mid C_j \), i.e., all connection services provided by connectors are used by components; and

3). \( (P_1 \mid \cdots \mid P_n \mid C_1 \mid \cdots \mid C_n)_{\text{simplified}} \Rightarrow \emptyset \), i.e., components and connectors can establish well-established connections.
3.2. Architectural Goals

After architectural reference models for software architectures are defined, the architectural goals of software systems can further be defined based on whether software architectures of the systems simulate the behaviors of their architectural reference models.

Architectural goals are divided into two categories, i.e., structural goals and interaction behavioral goals.

3.2.1. Structural Goal

The structural goal of a software system implies the topological requirements that developers are pursuing while building the software system. In the expression of the structural goals of software systems, those interaction behaviors among components need not be considered.

Definition 11. Suppose that $SA$ is a pi-calculus process defined for describing the software architecture of a software system, $RM$ is its architectural reference model and $RM = P_1 | \cdots | P_n | C_1 | \cdots | C_m$. Then $SA$ is said to be able to achieve the structural goal ($SG$) of the software system if $SA$ weakly simulates $P_{1_{\text{interface}}} | \cdots | P_{n_{\text{interface}}} | C_1 | \cdots | C_m$.

I.e., $SG = SA WS P_{1_{\text{interface}}} | \cdots | P_{n_{\text{interface}}} | C_1 | \cdots | C_m$.                (11)

For example, in the fault-tolerant client/server system, no matter whether the primary server is alive or down due to failures, the system must always involve a server and some clients. Therefore, the structure of the system should weakly simulate $Client_{\text{interface}} | Server_{\text{interface}} | CS$ at all time. I.e., the system has the following structural goal:

$RM_{c/s} = Client | Server | CS$ and

$SG_{c/s} = SA_{c/s} WS Client_{\text{interface}} | Server_{\text{interface}} | CS$.              (12)

3.2.2. Interaction behavioral Goal

The interaction behavioral goal of a software system tells us how components involved in the software system should interact with one another after they establish connections through connectors. In other words, the interaction behaviors among components in the system should conform to specified interaction protocols.

Definition 12. Suppose that $RM$ is an architectural reference model for software architecture $SA$ and $RM = P_1 | \cdots | P_n | C_1 | \cdots | C_m$, in which $\{i_1, i_2, \ldots, i_h\}$ and $\{o_1, o_2, \ldots, o_k\}$ are the sets of input and output ports, respectively, defined in components $P_1, P_2, \ldots, P_n$, and $\{c_{i_1}, c_{i_2}, \ldots, c_{i_h}\}$ and $\{c_{o_1}, c_{o_2}, \ldots, c_{o_k}\}$ are the sets of ports, provided by connectors, for attaching those input/output ports defined in components. Then the interaction specified by the architectural reference model can be described as follows.

$RM_{\text{interaction}} = P_{1_{\text{spec}}} | \cdots | P_{n_{\text{spec}}} | C_{1_{\text{protocol}}} | \cdots | C_{m_{\text{protocol}}}$.              (13)

And $SA$ is said to be able to achieve its interaction behavioral goal ($IG$) if $SA$ weakly simulates
\[ RM_{interaction} \{ c_{-i_1}/i_1, \ldots, c_{-i_k}/i_k, c_{-o_1}/o_1, \ldots, c_{-o_k}/o_k \}, \]
in which the substitutions mean that components and connectors have been attached together.

\[ I.E., IG = SA \ WS \ RM_{interaction} \{ c_{-i_1}/i_1, \ldots, c_{-i_k}/i_k, c_{-o_1}/o_1, \ldots, c_{-o_k}/o_k \}. \quad (14) \]

Similarly for the client/server system, clients should always be able to request services from the primary (or backup) server, i.e., clients can always properly be connected to the server and behave in a way abiding by the interaction protocol specified in the connector. So the interaction behavioral goal of the system can be expressed as follows.

\[ IG_{c/s} = SA_{c/s} \ WS \ Client_{spec} \ | \ Server_{spec} \ | \ CS_{Protocol} \{ c_{-request}/c_{-reply}/s_{-request}/s_{-reply} \}. \quad (15) \]

### 3.2.2. Overall Architectural Goal

Then the architectural goal (\( AG \)) of a software system is to achieve both the structural goal and the interaction behavioral goal.

\[ I.E., AG = SG \land IG. \quad (16) \]

For different styles of software architectures, there may exist different architectural reference models correspondingly. While pursuing the achievements of the architectural goals, software systems should also be able to embody those generic (or style-specific) constraints on software architectures into the definitions of their architectural reference models.

For example, for client/server style based software architecture, any client should connect with a server through a connector. Concretely, suppose that \( C = (v \ rep, req) \rightarrow (rep, req) \mid C_{spec} \) is a client involved in the system, in which \( req \) is an output port and \( rep \) is an input port and the (primary/backup) server \( S = (v \ sreq, srep) \rightarrow (sreq, srep) \mid S_{spec} \). This constraint can be described as the evolution of the system as follows.

\[ SA_{c/s} \rightarrow \ SA_{c/s} \{ v_{/\{rep\}}/i_{/\{req\}} \mid \ldots \}. \quad (17) \]

It means that all ports occurring in the client will be connected to ports provided in the server, i.e., the client will properly be connected to the server.

In fact, this constraint can easily be satisfied once if the interaction behavioral goal of the client/server system is achieved.

For another example, in C2 style-based software architecture, for any connector \( C \), there is no such a connector \( C' \) that meets two conditions simultaneously: 1) its level is higher than \( C \)'s but 2) some of its top components are \( C \)'s bottom components.

Suppose that \( C \) and \( C' \) are C2 connectors and \( C' \) is higher than \( C \). Suppose again that \( P_1, P_2, \ldots, P_n \) are bottom
components of \( C \) and \( Q_1, Q_2, \ldots, Q_m \) are top components of \( C' \). There are two sub-reference models of the software architecture corresponding to the two connectors, \( i.e., RM_1 = P_1 \parallel P_2 \parallel \ldots \parallel P_n \parallel C \), and \( RM_2 = Q_1 \parallel Q_2 \parallel \ldots \parallel Q_m \parallel C' \).

Then the composition of these two sub-reference models should not be similar with any Component-based Composition of them, \( i.e., \)

\[
\neg \exists i:1 \leq i \leq n, j:1 \leq j \leq m \left( P_i = Q_j \right) \land \left( RM_1 \parallel RM_2 \right) \]  

(18)

4. Changes

Changes occurring in software architectures usually include (Oreizy, 1996):

1. Adding new components or connectors, or removing existing components or connectors;
2. Upgrading or replacing existing components with new components; and
3. Reconfiguring, \( i.e., \) changing the relationships between components and connectors, for instance, disconnecting a component from a connector and then re-connect it to another connector.

For the simplicity of defining changes to software architectures, all entities manipulated by architectural operations can be assumed as atomic components and atomic connectors. Then complex changes can be expressed as compositions of those atomic operations.

4.1. Addition

Adding components (or connectors) can be implemented directly via using parallel composition operator of the pi-calculus.

Suppose that \( SA \) is the pi-calculus process defined for a software architecture and \( P \) is a new component to be added into to \( SA \). Then the operation of adding a component can be expressed as follows.

\[
SA' = SA \parallel Add(P)  
\]

(19)

\[
Add(P) = P
\]

Where, \( SA' \) is the new software architecture derived from \( SA \) after \( P \) is added in.

Adding a connector can be described similarly like adding a component except that the parameter of \( Add(x) \) is a connector instead of a component.

4.1. Removal

Because there is no operator for removing processes in the pi-calculus, the operation of removing components or connectors is implemented via adding restricted negative processes. After a negative process is added in, it will interact with its positive process and the composition of them will reduce into inert process \( 0 \). Thus the positive process will be isolated from other processes. Meanwhile, the restriction operator will make the interactions between the negative and
the positive processes invisible; and then all actions executed by the positive process will be hided from observation. As the result, the (positive) process seems to be removed.

Suppose that $P_i$ is the component to be removed. The operation of removing $P_i$ can be expressed as follows.

$$S^i = S | \text{Remove}(P_i)$$

$$\text{Remove}(P_i) = (\neg P_i)_{\text{removed}}$$

Similarly, removing a connector can be expressed by substituting a component with a connector in the above description.

### 4.3. Replacement

Replacing a component is equal to removing an existing component and then adding a new component. So the operation of replacement can be treated as a composition of operations of removal and addition.

Suppose that $P_j$ is going to take the place of component $P_i$. The operation of replacing $P_i$ with $P_j$ can be expressed as follows.

$$S^i = S | \text{Replace}(P_i, P_j)$$

$$\text{Replace}(P_i, P_j) = \text{Remove}(P_i) | \text{Add}(P_j)$$

While new components or connectors are added, it need mainly care whether the added entities can properly be embedded into the software architectures. Differently, for the replacements of components (or connectors), it is necessary to check semantically whether the new components could completely take the place of the original components. In other words, only when the new components take actions (or behave) in the similar ways and realize the same functionalities as the original components can it be said the replacements are complete and can the software systems work smoothly as if the replacements do not take place.

In principle, the properties of components can be specified in terms of interface, semantics, and interaction protocol (Vallecillo et al., 2000). To reason about the semantic replaceability of two components, Zaremski and Wing (1997) pointed out that two matched components, i.e., semantically equivalent components, should be with the same interfaces and the consistent behavior specifications, i.e., the pre/post-conditions of the two components should be able to imply one another mutually.

Since the semantics of the pi-calculus process is donated by the evolutions (especially communications) of processes (Milner et al., 1992), the semantic replaceability of two components can be reasoned about on two aspects, i.e.,

1. The interfaces, i.e., the way to connect to connectors, and
2. The behavior specifications, i.e., the sequences of actions exposed by the components.

And further, two components are mutually replaceable if the two components can (weakly) simulate one other.
In many cases, when components are added, new added components may be incorporated with other existing components into complex components.

For example, suppose that new added component $P_j$ and existing component $P_i$ will form a new component $P_{ij}$ after $P_j$ is added in. Then the operation of adding a component can be regarded as a replacement operation that replaces $P_i$ with $P_{ij}$.

$$SA = SA | \text{Replace}(P_i, P_{ij})$$

$$P_{ij} = P_j | P_i$$

In the new software architecture, $P_{ij}$ is treated as a whole.

### 4.4. Reconfiguration

When connectors are defined, components supposed to connect to connectors have been known. Changing the relationships of connections between connectors and components will require redefining the connection services provided by connectors. For example, when component $P$ is going to disconnect from $C_1$ and then connect to $C_2$, $C_1$ should not provide connection services for $P$ any more on one hand and $C_2$ should provide new connection services for $P$ on the other hand.

In the definitions of previous operations, software architectures are always treated as undividable things. So the operations of reconfiguration are define as compositions of several replacements. Considering the above example again, the reconfiguration can be defined as an implementation of the composition of two replacement operations, i.e., using $C_1 | (P_{\text{interface}})^{\text{restricted}}$ to substitute $C_1$ whilst using $C_2 | \neg P_{\text{interface}}$ to substitute $C_2$. Then disconnecting $P$ from $C_1$ and then reconnecting to $C_2$ can be expressed as follows.

$$SA' = SA | \text{Reconfig}(P, C_1, C_2)$$

$$\text{Reconfig}(P, C_1, C_2) = \text{Replace}(C_1, C_2 | (P_{\text{interface}})^{\text{restricted}})$$

$$| \text{Replace}(C_2, C_2 | \neg P_{\text{interface}})$$

Note that the reconfiguration operation is not defined via just one replacement operation, e.g., replace $C_1$ directly with $C_2$, because $C_1$ may still be connecting other components and cannot be removed completely.

### 4.5. Cascade Change

While changes occurring in software architectures, they rarely happen independently. That is, changes often result in cascade reactions in order to ensure the integrities of software architectures. For example, when a new component is added in, new connectors often need to be defined to provide connection services for the component. And when an existing connector is removed, all connected components may have to be removed as well; otherwise, new connectors should be defined to provide connection services for those isolated components.
Specifically, for example in the client/server system, when the primary server is removed from the software architecture due to its failures and the backup server joins in to take the place of the primary server, new connector should be defined between the client and the backup server to replace the old connector between the client and the primary server.

Rigorously speaking, no matter how changes take place, dynamic software architectures should always be able to guarantee that those components still involved in the systems could work properly.

Suppose that $C$ is the change (e.g., addition, removal, etc.) to be made to the system’s architecture and $Cc$ is the cascade reactions resulted from the change. And suppose further when the software architecture evolves in response to the change and the cascade, component $P_i$, $P_i = P_{\text{interface}} | P_{\text{spec}} (i_1, \ldots , i_m, \alpha_1, \ldots , \alpha_m)$, which still remains in the software architecture, will be affected. Then the evolution of the software architecture should satisfy the following dynamic constraint provided that the software architecture will evolve into $RM'$ after making the changes.

$$RM' = RM | C | Cc$$

$$RM' \rightarrow^* RM'\{ \frac{1}{i_1}, \ldots , \frac{1}{i_m}, \frac{1}{\alpha_1}, \ldots , \frac{1}{\alpha_m} \}$$

It implies that the dynamic software architecture should ensure that the affected component could still behave properly in the evolved system.

For different style based software architectures, cascade reactions resulted from changes may be different. For instance, in the client/server style, when a component is added in, a new connector should be added to provide a connection for the component at the same time; whereas in the C2 style, it may need only redefining an existing connector by extending the connector to provide more connection services.

When investigating changes and their results concerning software architectures, people should have a clear view about what cascade actions may happen, how the completeness of software architectures can be guaranteed, and how the architectural goals can always be achieved. All of these problems are to be tackled by using autonomous agents in the following context.

5. Reasoning about Impacts of Changes

The impacts of changes on software architectures are generally reflected on two aspects, i.e., the topological structures and the behavior modes. For example, adding or removing components (or connectors) may result in the alterations of the topological structures of original software architectures. Meanwhile, if a new added component adopts a different interaction protocol from that specified by existing connectors, the protocol specified by connectors may have to be redefined in order to ensure that connections between components and connectors are kept well-established. Thus, the behavior mode of the overall software architecture may alter.
However, no matter how a software architecture changes, it should satisfy the following requirements.

1). After changes take place, the software architecture can still maintain its completeness, i.e., all components and connectors are involved in well-established connections.

2). The result software architecture can meet the expected demand, i.e., can achieve the new desired architectural goals.

**Definition 13.** Suppose that $SA$ is a software architecture, $C$ is a change putting on $SA$, and $C'$ is the cascade reactions caused by $C$. Then the result of the change can be defined as a new software architecture provided that $SA \mid C \mid C'$ can form a complete software architecture.

$SA' = SA \mid C \mid C'$. \hfill (25)

For example, the software architecture of the client/server system can be defined as follows when the primary server is alive.

$SA_{c/s} = Client \mid PrimaryServer \mid CS_{pri}$ \hfill (26)

When the primary server is down and removed, the cascade reactions caused by the removal include starting the backup server to take the place of the failed primary server and replacing the connector between the client and the primary server by a connector between the client and the backup server. The backup server and the new connector can be described as follows respectively.

\[
BakServer = (v \_bs_{\_req}, bs_{\_rep}) Bakserver(bs_{\_req}, bs_{\_rep}) \mid BakServer_{spec}
\]

\[
CS_{Bak} = client(ci, co) \mid bakserver(si, so) \mid CS_{protocol}
\]

Then the change and its cascade can be described as follows.

$C = Remove(PrimaryServer)$ \hfill (28)

$C' = Add(BakServer), Reconfig(Client, CS_{pri}, CS_{buk})$

And the result software architecture is expressed as follows.

$SA'_{c/s} = Client \mid PrimaryServer \mid CS_{pri} \mid C \mid C'$ \hfill (29)

$= Client \mid BakServer \mid CS_{buk}$

As the architectural reference model, $RM$ can naturally achieve the architectural goal of software architectures, so if the concrete software architecture can weakly simulate $RM$, it will be able to assert that the software architecture achieves the architectural goal.

**Definition 14.** Suppose that $SA$ will change into $SA'$, and $RM$ is the architectural reference model defined for the desired software architecture. Then the result architecture is a desired software architecture if $SA'$ can achieve both the structural goals the interaction behavioral goals of the reference model, i.e., $SA' WS RM$. \hfill \Box

For example, the $RM$ for the client/server system is defined as follows.
Obviously, the software architecture $SA_{c/s}$ at the situation when the primary server is alive can weakly simulate the $RM$, i.e., $SA_{c/s} \overset{WS} {\rightarrow} RM$.

When the primary server is down due to its failures, the software architecture will evolve into $SA_{c/s}'$. Because $\text{BakServer WS Server}$ and $CS_{bak} WS CS$, it is reasonable to assert that $SA_{c/s}' \overset{WS} {\rightarrow} RM$, i.e., the result architecture $SA_{c/s}'$ is the desired software architecture of the client/server system.

### 6. Autonomous Agent-based Adaptation

To implement the automated adaptation to dynamic software architectures, an approach based on autonomous agents is adopted. In the approach, the reference model of the software architecture of a software system is defined as a local world model of an autonomous agent.

The simplified architecture of an autonomous agent can be shown in Fig. 5.

Within the architecture, there are several key elements (or components).

1). The architectural reference model defines the reference model of the software system.

2). The sensor is to perceive changes concerning the software architecture.

3). The planner has got two aspects of responsibilities.

   a). The planner acts as a manager of the architectural reference model and the desired architecture of the software system. As a manager, the planner will be responsible for reasoning about the architectural goals of the system and plan to achieve the goals. The plan for achieving goals is mainly to determine what components are needed, how components are connected, and what protocols are used for interactions.
b). The planner possesses some architectural style specific knowledge and behavior rules based on its knowledge to regulate its autonomous behaviors. When changes take place, the planner will plan the evolution of the software architecture according to those behavior rules on one hand, and automatically reason about the impacts of changes on software architectures and generate the destination software architecture on the other hand.

4). The component manager will manage addition and removal of components.

5). The connector manager is responsible for adding and removing connectors. And

6). The effecter is responsible for executing plans for achieving goals and for adjusting the software architectures.

The sensor monitors the run of the software system and perceives the changes to the software architecture. Once the sensor detects a change to the architecture or a change request from the software environment, it will transmit the change request to the planner. The planner reasons about the change and its cascade reactions based on its architectural style-specific knowledge and behavior rules and then makes a plan to adjust the software architecture. The plan will be dispatched from the planner to the component manager and the connector manager. Then the managers will define new or remove old components or connectors as request. And at last, the effecter will execute the plan and make changes to the software architecture, for instance, adding new defined components or connectors into the software architecture.

To make the description of the implementation more understandable, the following context will continue to use the example of the fault-tolerant client/server system to illustrate how the autonomous agents self-adapt to dynamic software architectures.

6.1. Architectural Style-specific Knowledge

For each architectural style, the knowledge is mainly concerned with what kind of components and connectors are involved in the style, what protocols those components use to interact in the way specified by the style, and what constraints the architecture should satisfy and maintain. Concretely, for each component, the knowledge is about the interface of the component and the protocol of the component’s behavior. For each connector, the knowledge is about how the connector maps its ports to components’ ports and acts as an interaction protocol adaptor to provide consistent connections for components.

For example, for a client/server style, there are clients and servers and the interactions between clients and servers are often based on request/reply protocols. Meanwhile, the architecture should satisfy some constraints such as a client must be connected to a server through a connector and a connector should provide connection services for and only for a client and a server.

In the implementation, the autonomous agent mainly uses its architectural style-specific knowledge to define or
create new connectors. Generally speaking, it is impossible for users to pre-define connectors for dynamic software architectures since users could not even know what components will be involved in software systems at runtime. So to be able to automatically adapt to dynamic software architectures, that autonomous agent should possess the capability of generating new connectors automatically just based on the interface and behavior specifications of components. Because of the space limitation of the paper, how to automatically generate connectors for two or more components can be referenced to the work in (Jiao and Mei, 2003).

6.2. Behavior Rules for Changes

As mentioned above, when the software architecture changes, the autonomous agent will plan the evolution of the dynamic software architecture. To make automated planning possible, a set of behavior rules for the agent are defined based on the architectural reference model. For every component (or connector) involved in the architectural model, each kind of changes is associated with a behavior rule. Every behavior rule is expressed as ‘situation $\rightarrow$ actions’, in which situation is in fact an architectural operation relevant to a component (or connector) to represent that a change takes place and actions are cascade reactions caused by the change. In principle, behavior rules are defined to guarantee that the current software architecture could successfully evolve into the desired software architecture and the evolving software architecture could always weakly simulate the architectural reference model.

In the previous section, there described four types of changes during the evolution of dynamic software architectures, among which the first three types of changes are relevant to components or connectors separately but the last one concerns components and connectors simultaneously.

For example, for a client $C$, there may exist four behavior rules in response to the changes relevant to the client.

- If $Add(C)$ then add a new connector to correlate the client with the server, to which server is determined by which server (the primary or the backup) is present currently.

  \[Add(C) \rightarrow Add(CS_{server})\]  

  where $CS_{server}$ is the connector for connecting the client and the server.

- If $Remove(C)$ then remove the connector for connecting the client and the server, too.

  \[Remove(C) \rightarrow Remove(CS_{server}).\]  

- If $Replace(C, C')$, i.e., replaced by another client, then remove the connector related to $C$ and add a new connector for $C'$ to correlate with the server.

  \[Replace(C, C') \rightarrow Remove(CS_{server}); Add(CS'_{server})\]  

  where $CS_{server}$ is the connector for connecting the old client and the server and $CS'_{server}$ is the connector for connecting the new client and the server.
• If \( \text{Reconfig}(C, \text{CS}_{\text{server}1}, \text{CS}_{\text{server}2}) \), i.e., disconnect the client from one server and reconnect it to another server (For example, switching from the primary server to the backup server, or in reverse.), then do nothing.

\[
\text{Reconfig}(C, \text{CS}_{\text{server}1}, \text{CS}_{\text{server}2}) \rightarrow \text{null}
\]  

(34)

For another example, for the primary server, if the system is not allowed to involve more than one server and the primary server can be replaced by (or replace) the backup server only when the primary server fails (or recovers), then when the primary server gets down due to failures, the desired software architecture would be

\[
\text{SA}_{c/s} = \text{Client} | \text{BakServer} | \text{CS}_{\text{bak}}
\]  

(35)

and when the primary server recovers from failures, the desired software architecture would be

\[
\text{SA}_{c/s} = \text{Client} | \text{PrimaryServer} | \text{CS}_{\text{pri}}.
\]  

(36)

Thus there may only exist two behavior rules relevant to the primary server.

• If \( \text{Remove} (\text{PrimayServer}) \), i.e., the primary server is down, then the backup server should be added in. Since the system is realized to provide fault-tolerance, the client should be reconnected to the backup server after the backup server takes the place of the primary server.

\[
\text{Remove} (\text{PrimayServer}) \rightarrow \text{Add}(\text{BakServer}); \text{Reconfig}(\text{Client}, \text{CS}_{\text{pri}}, \text{CS}_{\text{bak}})
\]  

(37)

where \( \text{CS}_{\text{pri}} \) defines the connector between the client and the primary server whereas \( \text{CS}_{\text{bak}} \) defines the connector between the client and the backup server. If there are more clients involved in the system, the \( \text{Reconfig} \) operation may repeat many times to reconnect all clients to the backup server.

• If \( \text{Add} (\text{PrimayServer}) \), i.e., the primary server recovers from failures, then remove the backup server and rebuilt connections for clients.

\[
\text{Add}(\text{PrimayServer}) \rightarrow \text{Remove}(\text{BakServer}); \text{Reconfig}(\text{Client}, \text{CS}_{\text{bak}}, \text{CS}_{\text{pri}})
\]  

(38)

Since the backup server is a passive entity involved in the system and it cannot autonomously join in or remove from the system, it need not define other behavior rules specific to the backup server.

In fact, for some changes, when people are not concerned with the cascade changes resulted from them, the behavior rules related to those changes can simply be expressed as \( \text{'situation} \rightarrow \text{null}' \), i.e., do nothing further when those changes take place.

### 6.3. Planning Evolution

The agent’s behavior rules mainly specify how the agent reacts to single changes. However, in many cases, changes may even take place simultaneously and they may be interdependent. For example, while a client is joining in, the primary server gets down. Obviously, the addition of the client and its cascade changes (i.e., building the connection with the server) depend on the replacement of the primary server. Concretely, if the agent reacts to the addition of the client first, it will build a new connector for connecting the new client to the primary server. Since the primary server
turns down nevertheless, the agent should reconfigure the software architecture via switching the client from the dead primary server to the backup server. That is to say, the agent has to replace the new generated connector immediately after the connector is created. This example manifests that the order of actions taken by the agent to react to changes, \textit{i.e.}, to handle the behavior rules, is very important for the agent to improve its performance.

In fact, that changes are interdependent is generally due to the interdependency of cascade reactions specified by behavior rules. For example, the reaction to the primary server’s removal may result in removing the connector between the client and the primary server, so any operations on that connector will be affected due to the removal of the primary server and furthermore those affected operations should be handled after the removal operation and its cascade changes finish; otherwise, those operations may be unavailing or have to be redone.

The main goal of planning is to generate the sequence of actions when one or more changes take place simultaneously. The result of planning should be able to support a good performance of evolution of the software architecture, \textit{i.e.}, the plan should avoid unavailing or superfluous actions as much as possible. For this goal, a precedence relation is defined among the behavior rules of the agent.

\textbf{Definition 9.} \textit{Precedence relation.} A behavior rule $R_1$ has precedence over another rule $R_2$ if 1) the right sides (\textit{i.e.}, the reactions) of the two rules refer to a common entity (component or connector) and 2) the common entity will be removed or replaced according to $R_1$ whilst it will be added according to $R_2$.

In other words, $R_2$ is restricted by $R_1$ if $R_1$ has precedence over $R_2$.

Obviously, the rules defined relevant to the primary server have precedence over those rules defined relevant to clients. The precedence relation can be depicted as follows.

\begin{figure}[h]
\centering
\includegraphics[width=0.8\textwidth]{precedence_relation.png}
\caption{Fig. 6. Precedence relation between behavior rules related to $\text{Remove(PrimaryServer)}$ and $\text{Add(C)}$}
\end{figure}

In Fig. 6, the reactions in response to the change of $\text{Remove(PrimaryServer)}$ involve three entities, in which the connector $CS_{pri}$ will be replaced by $CS_{bak}$. Meanwhile, the reaction cascaded by $\text{Add(C)}$ involves one entity, \textit{i.e.}, $CS_{pri}$ which will be removed in the reactions specified by the behavior rule related to $\text{Remove(PrimaryServer)}$. According to
the definition of precedence relation, the behavior rule related to Remove(PrimayServer) has precedence over that related to Add(C).

**Theorem 1.** Suppose there are changes related to a group of components or connectors. If the precedence relation among the behavior rules related to these changes is a *partial order*, then it is able to generate an optimal plan to handle all of these changes.

An optimal plan implies there are least unavailing or superfluous actions in the sequence of actions. To prove this theorem, it need first construct such a plan and then show that there is no plan better than the generated plan. The following algorithm can be used to construct the plan (see Fig. 7).

| **Input:** Changes. |
| **Output:** plan, i.e., the sequence of actions. |
| **Algorithm:** Constructing Plan |
| Initially, let the *plan* be a null sequence of actions, i.e., not including any actions. |
| 1) Obtain the *behavior rules* related to the changes; |
| 2) Analyze the *Precedence relation* among the behavior rules |
| 3) Repeatedly select a behavior rule that is not restricted by any other behavior rules until there is no such rules: |
| - Extend the plan via appending the cascade actions specified by the behavior rule. |
| 4) If there still exist behavior rules not selected by the last step, repeatedly select any one behavior rule *randomly* from them, and |
| - Extend the plan via appending the cascade actions specified by the behavior rules. |

**Fig. 7. Planning Algorithm**

Obviously, the algorithm will terminate. Since a partial order ensures that the succeeding actions would not cause the preceding actions to be unavailing or need to be redone, the plan generated by using the algorithm includes no unavailing or superfluous actions if the precedence relation is a *partial order*.

Considering the previous example again, i.e., the addition of a client and the failure of the primary server happen at the same time. Apparently, the behavior rule related to the withdrawal of the primary server has precedence over the rule related to the addition of the client, so the behavior rule in response to the removal of the primary server should be analyzed and handled before the rule relevant to the addition of the client. Therefore, the plan for coping with these changes will include the following actions orderly.

- *Add(BakSever)*; *Reconfig(Client, CS\textsubscript{pri}, CS\textsubscript{bak})*;
- *Add(CS\textsubscript{bak})*. Since the backup server has taken the place of the primary server, the new-added client need no longer be connected to primary server.
This example shows that if the agent does not plan the evolution, it may first build a relationship for connecting
the client to the primary server and then have to modify the relationship for redirecting the client to the backup server,
which implies that the agent may do more work than the necessity.

However, in some cases, there may not exist a partial order among the precedence relations of behavior rules \(i.e.,\)
the second case in the algorithm), the plan may have to include some unavailing and superfluous actions, which will
cause the agent to do some unnecessary work.

In principle, a feasible evolution plan should guarantee that the software architecture is always consistent with its
reference model.

**Definition 10.** Formally, suppose a plan \(P\) is a sequence of actions \(i.e.,\) operations of changes) and \(RM\) is the
architectural reference model. Then the plan is feasible for evolving the dynamic software architecture if the result
software architecture still weakly simulates the reference model, \(i.e.,\) \(P|_{RM} WS_{RM}\).

For the fault-tolerant client/server system, the evolution of the dynamic software architecture should guarantee that
the software architecture could smoothly switch from \(SA_{c/s}\) to \(SA'_{c/s}\) when the primary server fails and switch back from
\(SA'_{c/s}\) to \(SA_{c/s}\) when the primary server recovers.

It is obvious that the plan described above is a feasible one and also that the behavior rules defined for the agent
can guarantee the software architecture of the fault-tolerant client/server system to switch between different situations
smoothly.

6.4. Adaptation to Dynamic Software Architecture at Runtime

Conceptually, cascade reactions are actually to make changes to the dynamic software architecture. Changes made
by cascade reactions may result in further cascade reactions; and extremely, the process that cascade changes cause
further cascade reactions may be infinite. Therefore, while planning the evolutions of dynamic software architectures,
the autonomous agent need not consider the situations that cascade changes \(i.e.,\) changes resulted from cascade
reactions) may cause further cascade reactions. The implementation adopts such planning strategy, \(i.e.,\) planning just on
current changes and their cascade reactions, is because 1) the planning process may not terminate and 2) more
importantly, while the autonomous agent is planning the evolutions of dynamic software architecture, more changes
may occur and it is impossible for the agent to handle all changes at one stoke.

Since the executions of the evolution plans are actually making changes to dynamic software architectures, it is not
necessary to distinguish whether changes are newly arisen or resulted from the executions of the plans. This
consideration will make the adaptation to the dynamic software architecture at runtime easier.
Inside the autonomous agent, there is a sensor, whose responsibilities are to monitor the run of the software system and perceive the changes to the software architecture. By using the sensor, the agent can adapt to the dynamic software architecture at runtime as follows.

- While an evolution plan is being executed, perceives all changes to the software architecture, including the changes resulted from the execution of the plan;
- Once after the execution of a plan finishes, starts another round of planning on the newly perceived changes and executing plans.

In the above process, the cascade changes will be captured and handled in the next round of planning, so the agent can avoid getting into infinitely planning on infinite cascade reactions. Meanwhile, the agent can perceive and handle all changes to the dynamic software architecture at runtime and need not plan at all time.

6.5. Implementation of Autonomous Agent

The main parts of an autonomous agent are the sensor for perceiving the changes to the dynamic software architecture, the planner for planning the evolutions of the software architecture in response to the changes, and the effecter for executing plans to make cascade changes to the software architecture. So the description about the implementation of an autonomous agent will focus on these three parts.

By far, a prototype of the autonomous agent for self-adapting to dynamic software architectures has been implemented on a J2EE-compliant application server, named PKUAS (abbreviated for PeKing University Application Server), which is actually a component-based middleware platform and provides an environment for deploying and executing interoperable components (Wang et al., 2002; Huang et al., 2004).

In the PKUAS (see Fig. 8 for its componential infrastructure), containers provide runtime spaces for instances of components. To support the executions of instances of components, containers have the ability to capture all details (such as interfaces, constraints, and relationships) of components, which is realized via the reflection mechanism of the component-based middleware platform (i.e., the PKUAS).

![Fig. 8. Componential Infrastructure of the PKUAS](image)
Based on the reflection capabilities of containers, Huang et al. (2004) studied the means of recovering the runtime software architectures (RSAs) of software systems and implemented an interoperability framework on the PKUAS to support RSAs. Meanwhile, Wang et al. (2002) studied the evolutions of RSAs and put forward an online evolution mechanism for software systems at runtime. Based on these work and by taking the PKUAS as the supporting environment of agents for self-adaptations, autonomous agents are implemented on the PKUAS as follows.

- The sensor of an agent invokes the mechanism supporting RSAs to monitor the run of software systems and capture changes to software architectures; The RSAs provide the runtime views of components, connectors, and relationships between components and connectors and can further reason about the differences between the RSAs and the architectural reference models.
- The planner plans the evolutions in response to the changes perceived by the sensor, and then submits the evolution plans to the effecter for executions;
- The effecter then uses the online evolution mechanism to implement the executions of plans. By using the online evolution mechanism, the PKUAS can add/remove (or load/unload) components or connectors dynamically without needing to stop the runs of software systems.

7. Case Study

In the last section, the dynamic software architecture of a fault-tolerant client/server system has been studied via using the autonomous agent-based approach. However, the example fault-tolerant system seems a little too simple to some extends to illustrate the capacity of the approach proposed in this paper. So in this section, a more complicated example system with dynamic software architecture will be studied.

Consider a web-based management information system (MIS), in which web users access management services by making requests to geographically distributed servers via the MIS core (see Fig. 9).

![Fig. 9. Web-based Management Information System (MIS)](image-url)
In this system, the servers should be able to provide services with different qualities for different requirements of users. For example, a user who queries information is usually authenticated to read data maintained by the servers, and the MIS need not provide high quality services for the user, for instance, relatively lower security, longer request-response latency, and none transaction processing. On the contrarily, for an administration user who is responsible for manipulating and maintaining data, he is generally qualified to modify data on the servers. In this case, when the administration user requests services, the MIS should guarantee the qualities of services to be as higher as possible.

 Obviously, in the web environment, web clients and servers are not always online and connected with the MIS. Thus, the software architecture of the MIS may dynamically change due to the following reasons.

- Web users dynamically log on to (or log out from) the MIS. To serve web users, the MIS should dynamically create (or release) connections for web users.
- Web servers need not always be connected with the MIS and they are required to connect with the MIS and asked to provide services only when web users have requests. In other words, when web users log in and request for service, the MIS should first select appropriate servers and then establish connections for the selected servers. And when the selected servers finish providing services, the MIS may disconnect the servers for reducing costs.
- For different web users, they have different access right on one hand and they may request for service with different qualities of services on the other hand. To improve the performance and decrease the cost of serving, the MIS has the obligation to providing connections with different qualities of services for different users and servers instead of providing the same connections for all users and servers.
- A service may be provided by more than one server. However, the MIS should not make the loads of some server too heavy whilst others too light. So the MIS should dynamically select servers with fewer loads to serve web users. In addition, when some servers are out of services due to failures or overloaded, the MIS should promptly switch to other servers to provide highly reliable services.

The dynamic changes of the software architecture of the MIS mainly include:

- A web user logs in. When a new user logs in, the MIS should create a new connection for the user to connect to the MIS and according to the access right of the user, the new created connection will be assigned with corresponding quality of service.
- A web user sends out a new request. When a user requests for services, the MIS should first select an appropriate server and then connect the server with the MIS. Similarly, according to the request and the access right of the user, the connection between the MIS and the server should be with suitable quality of service.
A request is replied. After a server finishes providing service, the MIS can disconnect the server, i.e., remove the connection between the server and the MIS.

A web user logs out. When a user leaves the MIS, all relevant information (e.g., the connection between the user and the MIS) can be discarded from the MIS.

A server is disconnected or turns down while it is serving. While a server is serving, it had better stay online all the time so that the users could obtain services completely. However, due to the instability of the web and failures of the server itself, the server may be disconnected from the MIS and unavailable.

To self-adapt to the dynamic software architecture of the MIS, the MIS core integrates an autonomous agent which is implemented based on the agent-based approach described in the previous section. And meanwhile, to provide more reliable services, the MIS also implements a fault-tolerant mechanism to switch from a failed server to another server.

7.1. Architectural Goals

At different situations, the software architectures of the MIS may vary. Thus, for different situations, the architectural reference models of the MIS are varied. For example, when there are no web clients connecting to the MIS, i.e., no users have log on to the MIS, the architectural reference model of the MIS contains only one component, i.e.,

$$RM_1 = MIS$$

(39)

where MIS is the component denoting the MIS core.

When there are web users log on, the reference model of the MIS will include the MIS core and a set of users besides a group of connectors between the core and the users, i.e.,

$$RM_2 = Client \mid MIS \mid Conn_{cmis}$$

(40)

where Conn_{cmis} is a connector for connecting a client to the MIS core.

When web users request for service, the architecture of the MIS will include the MIS core, a set of users, and a collection of servers that can reply the requests. Naturally, the architecture should also contain connectors for linking the servers with the MIS. I.e.,

$$RM_3 = Client \mid MIS \mid Conn_{cmis} \mid Server \mid Conn_{cmis}$$

(41)

where Conn_{cmis} is a connector for connecting a server to the MIS core.

For distinguishing web users who request for service from those who just log on to the MIS without requesting services, a client can be defined as a composition of two components, one denotes the web user who log in and the other represents the request that the user has made. Thus, Client in the architectural reference models can be redefined as follows.

$$Client = Log-in-User \mid Request$$

(42)

In the definition, if the user just logs on without making any requests, client will only contain one part, i.e., Log-in-User.
The above architectural reference models describe the evolution process of the software architecture of the MIS, in which new components or connectors join in the MIS incrementally. Inversely, when requests are responded and users log out, the architectural reference models will degenerate gradually from \( RM_3 \) to \( RM_1 \). For example, when a server replies a request, the server will be disconnected from the MIS, which will make the architectural reference model degenerate from \( RM_3 \) to \( RM_2 \).

However, that a server is disconnected from the MIS may not always be because the server finishes performing a request. In some cases, a server may be isolated because of its failures or the errors of the communication links of the web, and it cannot be removed from the MIS without any remedy (That is why the MIS implements the fault-tolerant mechanism). Similarly as the definition of \textit{Client}, \textit{Server} can also be redefined as a composition of two parts, one denotes an active server and the other represents that the server is performing a request (\textit{i.e.}, providing a service).

\[
Server = ActiveServer \mid Service
\]  

(43)

Then, when the server is not providing a service or has finished providing a service, the server will contain only one part, \textit{i.e.}, \textit{ActiveServer}. Thus, the isolations of a server will be handled in different ways according to the reasons of the isolations (see the behavior rules (47) and (48) below).

In summary, the architectural goal of the MIS can be expressed as follows.

\[
AG_{mis} = S_{A_{mis}} \circ WS \ (RM_1 + RM_2 + RM_3)
\]  

(44)

where \( S_{A_{mis}} \) is the software architecture of the MIS.

### 7.2. Behavior Rules for Changes

In the above context, a list of dynamic changes that may occur in the MIS have been mentioned. According to those changes and the architectural goal of the MIS, a group of behavior rules can be defined and these behavior rules will be used to guarantee the architectural goal of the MIS to be achievable.

- When a new user logs in, the MIS should provide a connection for the user to connect to the MIS.

\[
Add(\text{Log-in-User}) \rightarrow Add(\text{Conn}_{mis})
\]  

(45)

- When a user makes a request, the MIS should find and activate a server to respond the request. Obviously, if the user has not logged in yet, this behavior rule will be unusable.

\[
Add(\text{Request}) \rightarrow Add(\text{Server}); Add(\text{Conn}_{mis})
\]  

(46)

where, \textit{Server} is composed of two parts, \textit{i.e.}, the server itself and a service for responding the request.

- When a server turns down due to its failures while it is responding a request, the MIS should switch to another server that can take the place of the failed server to keeping on providing the service.

\[
Remove(\text{Server}) \rightarrow Remove(\text{Conn}_{mis}); Add(\text{Request})
\]  

(47)

This behavior rule implies that the MIS will reconnect a new server via re-making the request.
• After a request is replied, the server should be released from the MIS.

\[ \text{Remove} (\text{Request}) \rightarrow \text{Remove} (\text{Service}); \text{Remove} (\text{ActiveServer}); \text{Remove} (\text{Conn}_{\text{mis}}) \]  

(48)

In this behavior rule, the server is not removed as a whole so that this rule will not trigger rule (47).

• When a user logs out from the MIS, the connection between the user and the MIS should be released and meanwhile the request made by the user should be discarded.

\[ \text{Remove} (\text{Log-in-User}) \rightarrow \text{Remove} (\text{Request}); \text{Remove} (\text{Conn}_{\text{mis}}) \]  

(49)

• Ideally, when a client is disconnected, the MIS should re-establish a connection for the client in order to improve the reliability of the system. However, it is usually difficult for the MIS to reason about whether the disconnection is because of the instability of the web or the fault of the client itself. So the MIS copes with the disconnection via considering simply that the client logs out from the MIS.

\[ \text{Remove} (\text{Conn}_{\text{mis}}) \rightarrow \text{Remove} (\text{Log-in-User}) \]  

(50)

• Similarly, when a server is disconnected, the MIS has to consider that the server turns down. Thus the behavior rule for this situation can be expressed as the same as rule (47).

7.3. Adaptation to Architectural Changes

Via using the behavior rules defined above, the MIS can adapt to most of dynamic changes relevant to the software architecture of the system.

For example, when the connection between a user and the MIS loses while the user is requesting for service, the MIS will trigger the behavior rule (50).

**Step1.** \( \text{Remove} (\text{Conn}_{\text{mis}}) \rightarrow \text{Remove} (\text{Log-in-User}) \)

Since the loss of the connection will result in a cascade action of removing the user, the cascade action will cause further cascade actions as specified by rule (49).

**Step2.** \( \text{Remove} (\text{Log-in-User}) \rightarrow \text{Remove} (\text{Request}); \text{Remove} (\text{Conn}_{\text{mis}}) \)

Furthermore, the execution of rule (49) will lead rule (48) to be triggered.

**Step3.** \( \text{Remove} (\text{Request}) \rightarrow \text{Remove} (\text{Service}); \text{Remove} (\text{ActiveServer}); \text{Remove} (\text{Conn}_{\text{mis}}) \)

In step 2, the action \( \text{Remove} (\text{Conn}_{\text{mis}}) \) may re-trigger rule (50). However, because the user has been removed at step 1 and the MIS could not remove the user once again, \( \text{Remove} (\text{Conn}_{\text{mis}}) \) will not result in new cascade actions any more.

In this example, if the user just logs in without making any requests, the execution of rule (49), i.e., step 2, will not cause rule (48) to be triggered.

In many cases, dynamic changes may occur simultaneously. To cope with multiple changes consistently, the MIS should coordinate the executions of behavior rules. As described in the above section, the MIS will first make a plan on
those multiple changes by analyzing the precedence relation among related behavior rules and then execute the plan to adapt to the dynamic software architecture of the MIS.

For example, while a user is requesting for service, links for connecting the user, the server to the MIS are broken. At this situation, the MIS can trigger both of rule (47) and rule (50). However, if the MIS executes rule (47) first, the new connected server will be removed immediately after rule (50) is triggered. So to cope with this situation, the MIS should plan the changes and the cascade reactions first and make an appropriate plan for changes. Through analyzing the precedence relation between rule (47) and (50), it is not difficult for the MIS to make a plan to execute rule (50) first and rule (47) afterward since rule (50) has precedence over rule (47).

8. Related Work

Currently, studies on dynamic software architectures are mainly at two levels. At the first level, how to understand and describe dynamic software architectures and changes occurring in architectures are investigated (Van der Westhuizen and van der Hoek, 2002; Wermelinger and Fiadeiro, 2000). And at the second level, how to develop dynamic software architecture based approaches for self-healing systems or self-adapting to changes are exploited (Schmerl and Garlan, 2002; Dashofy et al., 2002).

In (Dashofy et al., 2002), a strategy for effecting repairs in running software systems is proposed for architecture-based evolution. To reason about the validation of the evolution, this approach needs pre-defining both the “before” architecture and the “after” architecture and then verifies whether the changes could make the software system evolve from the “before” to the “after”. When the changes are verified to be valid, the approach will plan making repairs after the system is developed and deployed. However, this approach is mainly applied to event-based architectures and neither the reasoning process for validating changes nor the planning process for making repairs is presented in the literature.

In (Cheng et al., 2002; Garlan and Schmerl, 2002), an architecture-based approach is proposed for software adaptation, which is based on a three-layer architecture: 1) the runtime layer includes a monitoring infrastructure and captures information relevant to the application, 2) the model layer consists of an architectural model of the system and can adapt the application using a repair handler, and 3) the task layer is responsible for setting overall system objectives. In their approach, architectural styles are constraints over architectures and repairs are handled when constraints are violated. To repair software systems, repair operations should be pre-specified in the system code.

In (Huang et al., 2004), runtime software architectures are investigated based on the reflective mechanism of component frameworks and the adaptations of runtime software architectures are implemented on a middleware platform compliant with the J2EE specification.
While much work on self-adaptations is based on architecture, Karsai and Sztipanovits (1999) describe a domain-specific, multiple-view model-based approach to self-adaptive software. In model-integrated approach, self-adaptation is decomposed into two major issues: the issue of representation and the issue of the reconfiguration mechanism. The first deals with modeling self-adaptive systems and the second maps the models into executable systems and changes the application’s dataflow and control structure in a safe, consistent manner.

9. Conclusions

However, software architectures are always treated as constraints over software systems on one hand. On the other hand, most of current researches are ad hoc and cannot rigorously reason about the result of changes related to software architectures. Though some of them can cope with changes and then self-heal systems automatically, they are usually only fitted for some restricted situations such as with a fixed software architecture style.

In this paper, a new approach is adopted to study and implement the adaptations to dynamic software architectures of software systems.

First, software architectures are treated as goals that software systems are pursuing to achieve while software systems are designed and deployed. For defining those architectural goals, the notation of negative process is introduced to define the concept of architectural reference models. Thus, a software system is said to have achieved its architectural goals if its architecture could simulate its reference model on the aspects of topological structure and interaction behavior.

Second, software architectures and changes are described in a uniform formalism so that the result of changes can be reasoned about automatically. Traditionally, in the definitions of operations for changes, changes are often coped with independently in the literature. Unlikely, this paper studied the cascade reactions caused by changes and defined behavior rules to specify cascade reactions based on the architectural reference models so that the achievements of architectural goals can always be ensured.

Third, autonomous agents are used for planning the changes to software architectures. By using the behavior rules, autonomous agents can automatically plan to adjust software architectures and guarantee that the software architectures are always kept complete and the architectural goals can be achieved. The example described in this paper shows that, by using autonomous agents, software systems can easily adapt themselves to new situations.

With the widespread use and popularization of the Internet, software systems running on the Internet are naturally becoming more and more dynamic and autonomous and their software architectures are increasing dynamism and flexibility. By using goal-oriented autonomous agents to manage the adaptations of internet-based software systems to dynamic software architectures will provide a better solution for internet-based systems to autonomously adapt to dynamic changes.
Acknowledgements

I would like to thank the anonymous referees, whose useful comments have greatly improved the presentation of this paper.

This work is partially sponsored by the National Grand Fundamental Research 973 Program of China (2002CB31200003), the National Natural Science Foundation of China (60303004, 60233010, 60125206, 60103001), the National High-Tech Research and Development Plan of China (2001AA113060), and the Major Project of Science and Technology Research of Ministry Of Education P.R.C. (MAJOR0214).

References


Vitae

Wenpin Jiao. He got his B.E and M.E in Computer Software at East China University of Science and Technology in 1991 and 1997 respectively, and his Ph.D in Computer Science at Chinese Academy of Sciences in 2000. After he
finished his Ph.D study, he went to University of Victoria, Canada to be as a Post-Doctoral Fellow from 2000 to 2002. And now he is working at the School of Electronics Engineering and Computer Science of Peking University, China. His research interests mainly include software engineering, multi-agent systems, automatic computing, and formal methods.

Hong Mei. He received a Bachelor’s degree and a Master’s degree in Computer Science from Nanjing University of Aeronautics & Astronautics (NUAA) in 1984 and 1987 respectively, and a Doctorate degree in Computer Science from Shanghai Jiao Tong University in 1992. From 1987 to 1989, he was working at NUAA as a research assistant. In 1992, he joined in Peking University (PKU) as a post-doctoral research fellow. From 1994 to present, he was working at the Department of Computer Science and Technology and the School of Electronics Engineering and Computer Science in PKU, and became associate professor in 1994 and full professor in 1997. His current research interests include: Software Engineering and Software Engineering Environment, Software Reuse and Software Component Technology, Distributed Object Technology, Software Production Technology, and Programming Language.