Architectural Mismatch Detection between Component and Aspect Based on Finite Automata

Yang Zhang  
School of Information Science & Engineering  
Hebei University of Science & Technology  
Shijiazhuang, Hebei, P.R. China  
zhangyang@hebust.edu.cn

Jingjun Zhang  
Scientific Research Office  
Hebei University of Engineering  
Handan, Hebei, P.R. China  
santt88@163.com

Hui Li  
Scientific Research Office  
Hebei University of Engineering  
Handan, Hebei, P.R. China  
lihuiflying1981@163.com

Abstract—Architectural mismatch increases the underlying danger of compositional system and reduces the reusability of component. Traditional architectural mismatch throws much concern on the mismatch between components. Nonfunctional property, regard as the second or even third-class entity, is used to guide to choose component and connector, implement analysis of the performance and check the constraint. By introducing Aspect to software architecture, this paper extends the basic elements of software architecture by two means: 1) taken nonfunctional property as a first-class entity and 2) describing it with Aspect. Firstly, this paper defines the connection between component and aspect. The different way of composition between component and aspect determines the mismatch which is different from the mismatch between components. Secondly, the architectural mismatch is described through finite automata. Finally, the algorithm of architectural mismatch detection is proposed. A simple example validates the algorithm roughly, and the advantages as well as the problems of the algorithm are discussed.

Keywords—architecture mismatch, component, aspect, finite automata

I. INTRODUCTION

The notion of architectural mismatch is generally used to refer to incompatibilities that occur when assembling new software system from existing components which need not have common architectural origin [1, 2]. Architecture mismatch occurs when the properties of one component conflict with the properties of another one. Architectural mismatch increases the danger of compositional system and reduces the reusability of component. Research on architectural mismatch is benefit to non only ensure the correct design of software system, but also increase the reusability of the component.

The origin of architectural mismatch can be traced to the work of Parnas [1] on the effects of change on software design. The term Architectural Mismatch was coined much later by Garlan et al [2]. They identified four main categories of assumptions that can contribute to architectural mismatch: nature of components, nature of connectors, global architectural structure and construction process. A finer categorization of architecture mismatch is suggested in [3].

Architectural mismatch is sometimes characterized as being structural or behavioral [3]. By structural mismatch they mean static incompatibility of two components, where at least one of the components lacks appropriate code that would avoid mismatch. In contrast, behavioral mismatch implies a dynamic incompatibility, which shows up in a certain run-time environment that prevents the components from executing appropriate paths in their otherwise compatible code. Deadlock is the most common manifestation of the later form of mismatch. Orlandic et al [3] have performed their research work from the structural mismatch between components.

Software architecture depicts the whole structure of software, which plays an important role in the process of software development. Research on software architecture is benefit to discover the commonness of different software system, ensure flexible and correct design, and manage the global property. Traditional software architecture includes three basic elements: component, connector and constraint. Architectural mismatch occurs when the implementation properties of one component conflict with the properties of another one. Architectural mismatch may lead to the poor design of software system.

The work of [4][5][6] put their concerns on the mismatch among components, because the component is the first class entity in the architecture. Nonfunctional property crosscuts the functional property and has not taken as a first-class entity to be modeled and described. The mismatch between functional part and nonfunctional parts is not concerned about, mainly for the reason that nonfunctional spreads all over the inner software and isn’t modeled as a first-class entity.

Aspect-oriented programming (AOP) [7] provides a mechanism to modularize the crosscutting concerns. Introducing AOP to software architecture, this paper extends the basic elements of software architecture by considering nonfunctional property as a first-class entity and describing it with aspect. The extended software architecture includes four parts: component, aspect, connector and constraint. In the architecture, there exists the mismatch not only among components but also between component and aspect as component interacts with aspect [10]. Though much work has emerged about the mismatch among the components, the problem about the mismatch between component and aspect remains unresolved.

In this paper, we concentrate solely on the behavioral forms of mismatch. Firstly, we define the connection between component and aspect formally. Secondly, the architectural

978-1-4244-1676-9/08 ©2008 IEEE
mismatch between component and aspect is described through finite automata and the algorithm of architectural mismatch detection is proposed. Finally, we discuss the advantage and some problem of the algorithm.

II. COMBINATION BETWEEN COMPONENT AND ASPECT

Introducing AOP to architecture, the basic element of architecture is extended to four parts including of component, aspect, connection and constraint. In this section, we defined the model of component and aspect separately at first. Based on model of component and aspect, the static and dynamic combination between component and aspect are described.

**Definition 1.** A component \( C = (P, M, \beta) \), where \( P \) is a set of port, \( M \) is a set of message on the port and \( \beta \) is a mapping:
\[
\beta: P \rightarrow \{-1, +1\}
\]

For \( \forall p \in P \), if \( \beta(p) = -1 \), \( p \) is the output port of the component; if \( \beta(p) = +1 \), \( p \) is the input port of the component.

The model of component that is divided into input port and output port. The input port receives the message from the aspect and other components while the output port sends out the message.

**Definition 2.** An aspect \( A = (Adv, M, \gamma) \), where \( Adv \) is a set of advice, \( M \) is a set of aspect message, \( \gamma \) is a mapping:
\[
\gamma: Adv \rightarrow \{A_k, A_c\}
\]

For \( \forall adv \in Adv \), if \( \gamma(adv) = A_k \), \( adv \) is the sink advice of aspect; if \( \gamma(adv) = A_c \), \( adv \) is the source advice of aspect.

In the model of aspect, an advice of an aspect is divided into sink advice and source advice that are prone to interact with the component.

The AOSD community offers different approaches for weaving aspects, depending on the points where the Pointcuts can be placed. Some approaches support the definition of Pointcuts at any place of the code (e.g., before, after, around, ...), mainly because they are based on the code intrusion. Different kinds of message interception are used in other approaches, so the aspect evaluation is triggered by the delivery of a message or an event. This allows aspects to be applied to black-box component, closely to the CBSD philosophy. The aspect is evaluated when intercepting the message that sent out from the component. The static combination between component and aspect describe the connection between the port of component and advice of aspect. The dynamic combination is expressed through the executable model of finite automata.

Supposed that there is component \( C = (P, M, \beta) \) and aspect \( A = (Adv, M, \gamma) \), the static connection between component and aspect is a mapping:

\[
\text{Apply: } P \times Adv \rightarrow \{0, 1\}
\]

For \( \forall a \in Adv \), \( p \in P \), if the mapping \( \text{Apply}(p, a) = 1 \), there exists the connection between component and aspect; if the mapping \( \text{Apply}(p, a) = 0 \), there doesn’t exist the connection between component and aspect.

For \( \text{Apply}(p, a) = 1 \), if \( \beta(p) = -1 \), \( \gamma(a) = A_c \); if \( \beta(p) = +1 \), \( \gamma(a) = A_k \).

The dynamic combination mainly defined through finite automata.

**Definition 3.** A finite automata is a quintuple \( FA=(S, \Sigma, f, S_0, Z) \), where \( S \) is a finite set of states, \( \Sigma \) is a finite set of input message alphabet, \( f \) is the mapping from \( S \times \Sigma \) to \( S \), \( S_0 \) is a set of initial state, \( Z \) is a set of final state.

Let \( \varepsilon \) be empty message, \( \varepsilon \in \Sigma \). Empty message has no impact on transition of state, then

\[
\begin{align*}
(1) & f'(s, \varepsilon) = s; \\
(2) & f'(s, b\varepsilon) = f'(f(s, b), w), \text{ where } b\in\Sigma, w\in\Sigma^*, f' \text{ is a mapping from } S \times (\Sigma \cup \{\varepsilon\}) \text{ to } 2^S.
\end{align*}
\]

Let finite automata \( FA \) is an implementation of a system that composed by component \( C_i(1 \leq i \leq n) \) and aspect \( A_j(1 \leq j \leq m) \), when \( \forall a \in Adv \) and \( p \in P \), \( \text{Apply}(p, a) = 1 \), we say \( FA_T = (S_T, \Sigma_T, f_T, S_{0T}, Z_T) \), where

\[
\begin{align*}
S_T &= S_1 \times S_2 \times \ldots \times S_{n+m} \\
\Sigma_T &= \Sigma_1 \times \Sigma_2 \times \ldots \times \Sigma_{n+m} \times \Sigma_1 \times \ldots \times \Sigma_n \times \Sigma_1 \times \ldots \times \Sigma_m, \text{ the message of component and aspect build up input alphabet of } \Sigma_T; \\
Z_T &= Z_1 \times Z_2 \times \ldots \times Z_{n+m}.
\end{align*}
\]

For \( \forall C_1, C_2 \in C \), a component \( C_1 = (P_1, M_1, \beta_1) \) and \( C_2 = (P_2, M_2, \beta_2) \), the port \( P_1 \) and \( P_2 \) connected, there exists aspect \( A \), let \( \text{Apply}(p_1, a_c) = 1 \wedge \text{Apply}(p_2, a_c) = 1 \), it shows that the message of component is intercepted by aspect \( A \). After aspect \( A \) deal with it, the message passes to \( C_2 \). Here \( a_c \) and \( a_k \) allow null. When \( a_c \) and \( a_k \) are null, it represents two component connected directly without the action of aspect.

III. ARCHITECTURE MISMATCH

A. Algorithm Proposed

Component and aspect co-exist in the software architecture. There may be mismatch between them if they are not properly organized. For example, when a component send out message \( a \) prior to message \( b \), a aspect needs to intercept the message and requires the message \( b \) prior to message \( a \). When this situation happens, the mismatch occurs. Another example of mismatch is that a component and an aspect visit the same critical resource. We define the architecture mismatch as following.

**Definition 4.** Let \( \varepsilon\)-Closure\((q) \) be a set of state if the following conditions are satisfied:

\[
(1) \quad \forall q \in S
\]

(2) The set is composed by some state that begin from \( q \), and \( \varepsilon \) is the input message

**Definition 5.** For \( \forall S_i \in S \), \( S_i \rightarrow S_j, S_i \rightarrow^* S_j \), \( \rightarrow \) is said to be the transition of a state.

**Definition 6.** Let \( S_i \) is the state of finite automata \( FA \), \( S_i \in S \), if there exists a finite state sequence that started from init state, \( S_0 \rightarrow S_1 \rightarrow S_2 \rightarrow \ldots \rightarrow S_{n-1} \rightarrow S_n \), \( S_0 \) is the init state and \( S_n = S_i \), we say \( S_i \) is reachable.
**Definition 7.** For \( \forall S_i \in S \), if \( S_i \) is reachable and \( S_i \in Z \), we say \( S_i \) is the reachable final state.

**Definition 8.** The system that composed by component and aspect exist the architecture mismatch, if and only if the following conditions are satisfied:

1. \( \forall a \in \Sigma, T = f'(\{s_{i1}, s_{i2}, \ldots, s_{im}\}, a) \);
2. \( q_i = \epsilon\text{-Closure}(T) \);
3. the state of set \( q_i \) is the unreachable final state.

Base on the definition above, the algorithm of architecture mismatch detecting is proposed as following.

**Begin**

\[
q_0 = \epsilon\text{-Closure}(S_0);
S_{new} = \{q_0\};
\]

For each unsigned \( q_i \) in \( S_{new} \)

\[
q_i = 1;
\]

For each of \( a \in \Sigma \) // \( \Sigma \) is composed by the message of component and aspect

\[
T = f'(\{s_{i1}, s_{i2}, \ldots, s_{im}\}, a);
q_i = \epsilon\text{-Closure}(T);
\]

**End For**

If \( q_i \notin S_{new} \)

put \( q_i \) into the set \( S_{new} \)

add the transition of state \( f'(q_i, a) \) to \( f' \)

**End If**

**End For**

For each of \( q_i \in S_{new} \)

If the element of set \( q_i \) not in the reachable final state set

Mismatch = true;

**End If**

**End For**

**End**

**B. An Example**

Taken the book management system that we developed as an example, the algorithm is validated roughly. In the software, two components and two aspects are identified: borrowing component and returning component, security aspect and logging aspect. The use of two components needs the validation of security aspect and the record of logging aspect. The security aspect is set prior to logging aspect, which means that only the legal user can log on the system and record the action of login. We analyze the action of component and aspect to get all the initial state of them, which is composed of \( q_0 \) and the reachable final state set.

The states of transition of borrowing component are as following:

\[
S_{10} \xrightarrow{\text{validation}} S_{11} \xrightarrow{\text{book inf}} S_{12} \xrightarrow{\text{registration}} S_{13} \xrightarrow{\text{book}} S_{14}. \text{The states of transition of returning component are as following: } S_{20} \xrightarrow{\text{validation}} S_{21} \xrightarrow{\text{book}} S_{22} \xrightarrow{\text{registerInf}} S_{23}
\]

\( \Sigma \) is composed by all the messages of component and aspect and adds all the middle state to \( q_i \) to create \( T \). The message of aspect and component, such as validation data, book data, etc can be constructed of \( a \). In this example, we set the message of message \( A \) is prior to message \( B \). Message \( A \) and \( B \) affect Message validation. Through the message transition, we get \( q_i \). The element of set \( q_i \) is not in the reachable final state set, and the mismatch is true.

**IV. RELATED WORK AND ALGORITHM DISCUSSION**

Software architecture provides a basis to the large software system. Introducing AOP to software architecture may achieve separation of concerns in a high level [8]. It helps to analyze the nonfunctional property and the relationship between component and aspect. Furthermore, combining AOP and software architecture is an effective way to apply and validate AOP in a large-scale software system.

Much work has been done on architecture mismatch. Compare et al [4] demonstrate that the use of formalism is an effective mechanism for detecting mismatch in dynamic behavior of existing components assembled into the Chemical Abstract Machine. Orlandic et al [3] perform their work on the architecture mismatch among components from structure of the component. They proposed a mismatch-free architecture to avoid the mismatch. The tool of Unicon [5] may provide mismatch detection, but it has the limitation in checking the global properties. Zhang et al [6] propose a behavioral mismatch description and the algorithm among components. Vanderperren et al [9] propose a visual component composition environment with advanced aspect separation features. But they don’t consider the mismatch between them. The works mentioned above put their concerns on the mismatch among components. However, the mismatch among components is different from the one between component and aspect as the difference of the interaction. So the algorithm aims to resolve the mismatch between component and aspect.

Now we have simply tested the algorithm through some system we developed. The result shows that it is useful for security aspect and logging aspect that are extracted from the software system. However, when concurrent aspect and real-time aspect are considered, the algorithm is not efficient for their feathers of aspects related with the time. Improvement needs to be done on our algorithm for concurrent aspect and real-time aspect, which will be our future work.

**V. CONCLUSION**

The innovation of this paper is that we research on the mismatch between component and aspect. We formally define the connection between component and aspect, and the architectural mismatch between component and aspect is described through finite automata and the algorithm of architectural mismatch detection is proposed. The algorithm is benefit to discover the mismatch in architecture.

The next work includes of the improvement our algorithm for concurrent aspect and real-time aspect, continuing to validate the algorithm and analysis the performance of the algorithm, such as the time and space complexity.

**ACKNOWLEDGMENT**

This work was supported by the Natural Science Foundation of Hebei Province under Grant No. F2006000647, P.R. China; Science-Technology Foundation of Hebei Province under Grant No. F2006000647.
REFERENCES


