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 not 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

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, β), where P is a set of port, M is a set of message on the port and β is a mapping:

β: P→{-1, +1}

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

The model of component defined the port 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, γ), where Adv is a set of advice, M is a set of aspect message, γ is a mapping:

γ: Adv→{A_k, A_c}

For $\forall$ adv∈Adv, if γ(adv) = A_k, adv is the sink advice of aspect; if γ(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, β) and aspect A = (Adv, M, γ), the static connection between component and aspect is a mapping:

Apply: P × Adv → {0, 1}

For $\forall$ a∈Adv, p∈P, if the mapping Apply(p, a)=1, there exists the connection between component and aspect; if the mapping Apply(p, a)=0, there doesn't exist the connection between component and aspect.

For Apply(p, a)=1, if β(p)= -1, γ(a)= A_c; if β(p)= +1, γ(a)= A_k.

The dynamic combination mainly defined through finite automata.

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

Let ε be empty message, ε ∈Σ. Empty message has no impact on transition of state, then

(1) *f*' (s, ε) = s;

(2) *f*' (s, bw) = *f*' (*f*(s, b), w), where b∈Σ, w∈Σ^*, *f*' is a mapping from S×(Σ∪ {ε}) to 2^S.

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

$S_T = S_1×S_2×…×S_{n+m}$

$\Sigma_T = \Sigma_{i1}×\Sigma_{i2}×…×\Sigma_{in}×\Sigma_{j1}×\Sigma_{j2}×…×\Sigma_{jm}$, the message of component and aspect build up input alphabet of Σ_T;

$Z_T = Z_1×Z_2×…×Z_{n+m}$

For $\forall$ C_1, C_2∈C, a component C_1= (P_1, M_1, β_1) and C_2= (P_2, M_2, β_2), the port P_1 and P_2 connected, there exists aspect A, let Apply(p_1, a_c)=1 ∧ Apply(p_2, a_k)=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. Algrorithm 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 ε-Closure(q) be a set of state if the following conditions are satisfied:

(1) $\forall$ q∈S

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

Definition 5. For $\forall$ S_i∈S, S_j∈S, $S_i \xrightarrow{m} S_j$, → is said to be the transition of a state.

Definition 6. Let S_t is the state of finite automata FA, S_t∈ S, if there exists a finite state sequence that started from init state, $S_0 \xrightarrow{m_0} S_1 \xrightarrow{m_1} S_2 \xrightarrow{m_2} \ldots \xrightarrow{m_{n-1}} S_n$, S_0 is the init state and S_n=S_t, we say S_t is reachable.

Definition 7. For $\forall S_t \in S$, if S_t is reachable and $S_t \in Z$, we say S_t 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) $q_j = \varepsilon\text{-Closure}(T)$

(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
$q_0 = \varepsilon\text{-Closure}(S_0)$;
$S_{new} = \{q_0\}$; // S_{new} is a new set of state
For each unsigned q_i in S_{new}
$q_i = 1$; // q_i is signed
For each of $a \in \Sigma$ // Σ is composed by the message of component and aspect
$T = f'(\{s_{i1}, s_{i2}, \ldots, s_{im}\}, a)$;
$q_j = \varepsilon\text{-Closure}(T)$;
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 to 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: $S_{10} \xrightarrow{validation} S_{11} \xrightarrow{book\ info} S_{12} \xrightarrow{registration} S_{13} \xrightarrow{book} S_{14}$. The states of transition of returning component are as following: $S_{20} \xrightarrow{validation} S_{21} \xrightarrow{book} S_{22} \xrightarrow{registInfo} S_{23}$

Σ 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. 07215601D-3, P.R. China; and the Scientific Research Foundation of Hebei University of Science and Technology under Grant No.XL2005063, P.R. China.

REFERENCES

[1] D.L. Parnas, "On the criteria to be used in decomposing systems into modules", Communications of ACM, Vol.15, No.12, pp. 1053-1058, 1972.

[2] D. Garlan, R. Allen, J. Ockerbloom, "Architectural mismatch or why it's so hard to build systems out of existing parts", In proceedings of 17th Int. Conf. on software engineering, Seattle, WA, pp. 179-185, 1995.

[3] R. Olandic, J.L. Pfaltz, "Prevent mismatch of homogenous components in the design of software architecture", International journal of software engineering and knowledge engineering, Vol.11, No.6, pp. 731-759, 2001.

[4] D. Compare, P. Inverardi, A.L. Wolf. Uncovering architecture mismatch in component behavior. Science of computer programming. Vol.33, No.2: pp. 101-131, 1999.

[5] M. Shaw, R. Deline, D.V. Klein, et al. Abstractions for software architecture and tools to support them. IEEE transactions on software engineering, Vol.21, No.4, pp. 315-335, 1995

[6] B. Zhang, K. Ding, J. Li. "An XML-message based architecture description language and architectural mismatch checking", In Proceedings of 25th Annual International Computer Software and Application Conference, Chicago, USA, 2001.

[7] G. Kiczales, "Aspect-oriented programming the fun has just begun", In Workshop on New Visions for Software Design and Productivity: Research and Applications, December 2001.

[8] M. Hong, D.G. Cao, "ABC-S2C: Enabling separation of crosscutting concerns in component-based software development", Chinese Journal of Computers, Vol.28, No.12, pp.2036-2044, 2005.

[9] W. Vanderperren, D. Suvée, B. Wydaeghe, V. Jonckers, "PacoSuite and JAsCo: A Visual Component Composition Environment with Advanced Aspect Separation Features", In proceedings 6th International Conference FASE 2003, April 7-11, 2003.

[10] J.J. Zhang, Y. Zhang, F.R. Li. "Combinatorial model and aspect-oriented extension of architecture description language". Proceedings of IEEE 3rd International Conference on Information Technology: Research and Education (ITRE'2005). Hsinchu, Taiwan. June 27-30, 2005, 277-281.