A Tool to Construct One Comprehensive Legal Environment for Behavioral Incompatible Components

Jizhou Zhao, Jiangwei Li, Yan Zhang, and Tian Zhang

[1] Department of Computer Science and Technology,
Nanjing University, Nanjing, P.R. China
{zjz,ljw}@seg.nju.edu.cn
ztluck@nju.edu.cn
[2] Department of Computer Science and Technology,
Beijing Electronic Science and Technology Institute, Beijing, P.R. China
zhangyan@besti.edu.cn

Abstract. Behavioral incompatibility in component compositions is an important problem in the field of component-based software development. To solve this problem, one approach is to construct an environment in which the incompatible components can work together. So we write this tool, which uses the interface automata to model the behavior of components, to derive available behaviors all out from two incompatible component compositions and construct a comprehensive legal environment for them. This paper presents all the details of our tool, including the main steps, the core algorithm and the analysis of efficiency. A case-study is also illustrated showing the validity and practicability of our tool.

1 Introduction

Component reuse is commonly known as a valid and practicable measure to efficiently develop high quality software at a low cost. By using components as reusable building blocks, we can rapidly and economically attain reliable, flexible, extensible and evolvable systems [6].

Component composition is one of the major problems of component-based software development. Behavioral incompatibility, related to the problem we concern in this paper, is one issue of component composition. The meaning of behavioral incompatibility is that between two composed components no message will ever be sent by one, whose reception hasn't been anticipated in the design of the other [4].

We developed this tool to solve the behavioral incompatibility problem, in the approach of constructing one environment for the incompatible components, in which they can work together without any error. In detail, firstly we compose two incompatible components together. Secondly, the valid transition set is derived. Thirdly, according to the valid transition set, we construct one comprehensive

legal environment, such that two incompatible components can work together and the behaviors of their composition can be preserved as much as possible.

We use interface automata [5] to model the behavior of components, and the products of the interface automata to represent the compositions of components, as the optimistic approach in the interface automata theory suits to our problem to be solved. The optimistic approach here, different from other formal methods, means that two components are seen as compatible if there exist an environment for them to work together without any error. Based on the optimistic approach, interface automata make the problem easier.

Although we have settled the behavioral incompatibility problem to some extent, this tool is not able to model the behavior of components into interface automata by itself. Likewise, the comprehensive legal environment we get from the tool cannot be mapped to the original component automatically. In other words, the support offered by our tool to derive available behaviors is limited to the operations on the interface automata modeling the behavior of original components. This tool is developed in the language of JAVA with Eclipse and has to run under the environment of JVM.

2 Overview

Our tool is consisted of four modules as followed: the Input Module, which allows user to import two interface automata. The Compose Module, which composes the two interface automata together. The Delete Module, which deletes the illegal states and derives the valid transition set. The Environment Module, which deals with the transitions of the valid transition set and constructs one comprehensive legal environment.

Only thing for user to do is to input their interface automata, the tool will handle the rest. Click the output button and the results are already there to be seen.

Figure 1 shows the interaction of four main modules. The arrows in the figure show how the data go.

3 Approach

In this section, most of concepts about interface automata refer to [5] and the main algorithm refers to [4].

We view the interface automata as a graph to simplify the operation and make it easier to be understood. To express the graph, we choose adjacency list as the main data structure. Each state of the interface automata corresponds to a vertex of the graph. Each transition of the interface automaton corresponds to an edge of the graph. The action of each transition corresponds to the weight of the related edge of the graph.

As is mentioned above, four modules build up the main part of our tool. In this section, we will discuss the details of Compose Module, Delete Module and Environment Module.

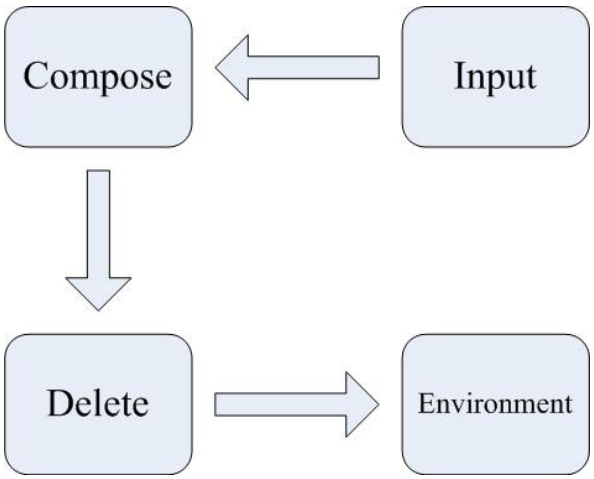

Fig. 1. the interaction of four main modules.

3.1 The Compose Module

After getting the two input automata, we can start to compose. Before that, whether the two automata can be composed or not should be taken into consideration. Two interface automata are composable only if they don't share the same input action or output action and any internal action of one automaton cannot be the action of another one. We write a method named "canBeComposed" to determine.

If the two interface automata are composable, the next step we do is to calculate their product, which seems to be the composed interface automata. However, the product is not the final result that we want. In order to facilitate the operation later, only states that can be reached from the initial state are valid. To pick out the valid states, we put the transitions of the product into an array-list. Then start from the initial state, we traverse the array-list through every possible path that can be reached. Those paths can be split into the valid transitions of the composed interface automaton.

With the valid transitions, we can rebuild the composed interface automaton in adjacency list. The states of the composed interface automaton are the states that valid transitions go through. The actions of the composed interface automaton are the actions of valid transitions plus the non-enabled actions of the original two interface automata.

3.2 The Delete Module

The Compose Module already gives out the composed interface automaton of the two input interface automata in the form of adjacency list. The next step is to derive the valid transition set. In the Delete Module, we will handle this problem.

First of all, we should try to find all the illegal states. Suppose there are two interface automata P and Q. In their composition PQ, illegal states are those at which one interface automaton cannot accept the input action provided by the other interface automaton. We write a method named "findIllegalStates" to do this job. As long as those illegal states exist, the two interface automata are behaviorally compatible. Our target in this module is to remove the illegal states.

The composed interface automaton is expressed by adjacency list. For the purpose of convenience, first we should transform adjacency list into inverse adjacency list. The method "list2inlist" is designed for the transaction. Then we start from one illegal state, along output actions and internal actions, find all the states that can be arrived. Delete those states as well as the path related. Keep repeating until there is no illegal state left. Use method "list2inlist" to transform the inverse adjacency list left into adjacency-list form, which is the valid transition set we need.

3.3 The Environment Module

Having derived the valid transition set, the last and the most complicated step is to deal with the transactions of the valid transition set and to construct the comprehensive legal environment.

If the valid state transition is empty, we can say that the comprehensive legal environment doesn't exist. Else, according to the algorithm referring to [4], there are four rules to construct the transitions of the comprehensive legal environment.

Suppose transition T starts from state X and ends in state Y along with the action A.

Rule 1: if X and Y is one same state, the corresponding transition of the comprehensive legal environment shall start from one state and end in the same state, too.

Rule 2: if there exists one path which starts from Y, ends in the initial state and each action is an internal action, then the corresponding state of Y shall be the initial state in the comprehensive legal environment.

Rule 3: if the corresponding state of Y in the comprehensive legal environment doesn't exist, create a new state and make it be the corresponding state of Y.

Rule 4: if the action A is an internal action, the corresponding transition shall be changed into one single state.

We traverse the transitions of the valid transition set in the way of depth-first and construct corresponding transition according to the rules mentioned above. All the corresponding transitions constitute the comprehensive legal environment.

4 A Case Study

This section presents a case study of the construction of two interface automata.

First we open the input interface and import two interface automata. Figure 2 shows the illustration of the interface automaton Comp and Figure 3 shows its adjacency list.

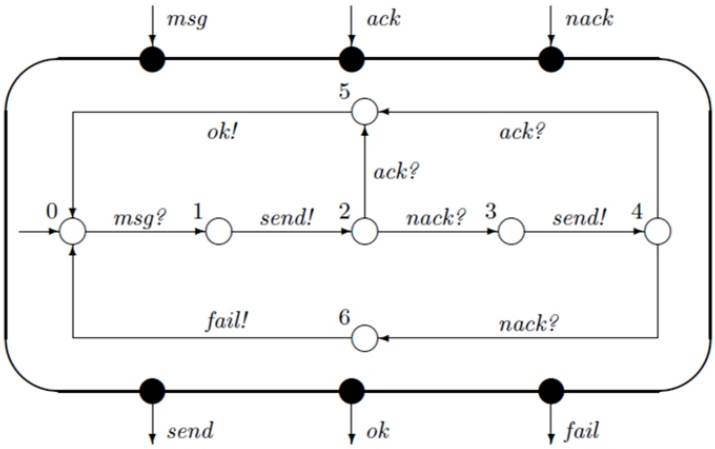

Fig. 2. the Comp in the IA form.

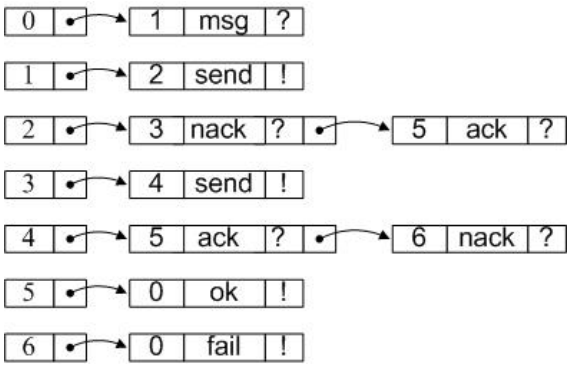

Fig. 3. the Comp in the adjacency-list form.

Suppose there are two sides named A and B. The interface automaton Comp describes a component behavior, which receives a message from side A (msg) and then sends the message to side B (send). If the first trial is failed, send again. After sending the message up to twice, a confirmation from B will be received. If the confirmation is "ack", which means B has received the message successfully, a new action "ok" will send information to A to announce the success. If the

confirmation is "nack", which means B fails to receive the message, a new action "fail" will send information to A to announce the failure.

Figure 4 shows the illustration of the interface automaton Client and Figure 5 shows its adjacency list.

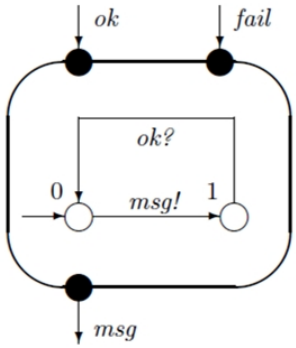

Fig. 4. the Client in the IA form.

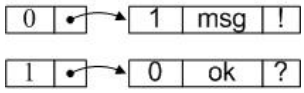

Fig. 5. the Client in the adjacency-list form.

The interface automaton Client uses the Comp to send messages to one side and always expects a successful transmission, which means the Client only deals with the information send by the "ok" action announcing success and ignores the other action. It is obvious that the action "fail" is not enabled on the state 1, so the two interface automata are behavioral incompatible.

After the operation of composition, we can get a new interface automaton. Figure 6 shows the illustration of the composed automaton and Figure 7 shows its adjacency list.

We can see that the state 6 is an illegal state, since the action "fail" is not enabled on the state 1 in the Client. In the valid state set, we delete the state 6 and the paths leading to it. Figure 8 shows the valid transition set and Figure 9 shows its adjacency list.

It is easy to tell that there exists a comprehensive legal environment for the composed interface automaton of the Client and the Comp as the valid transition set is not empty. So we can start to traverse the transitions of the valid transition set in the way of depth-first. The first transition is (0, msg, 1). According to rule

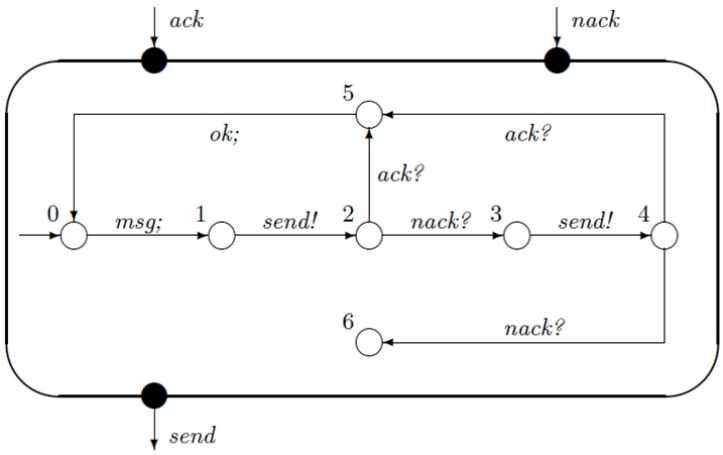

Fig. 6. the composition of Client and Comp in the IA form.

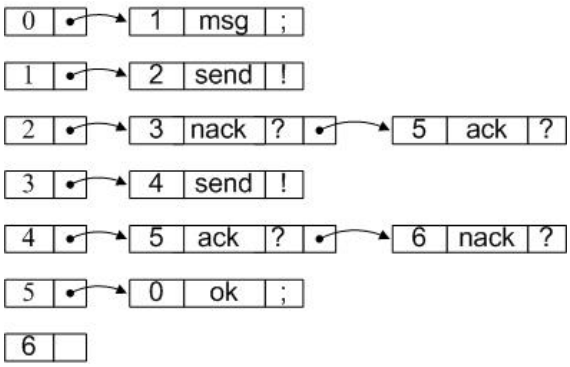

Fig. 7. the composition of Client and Comp in the adjacency-list form.

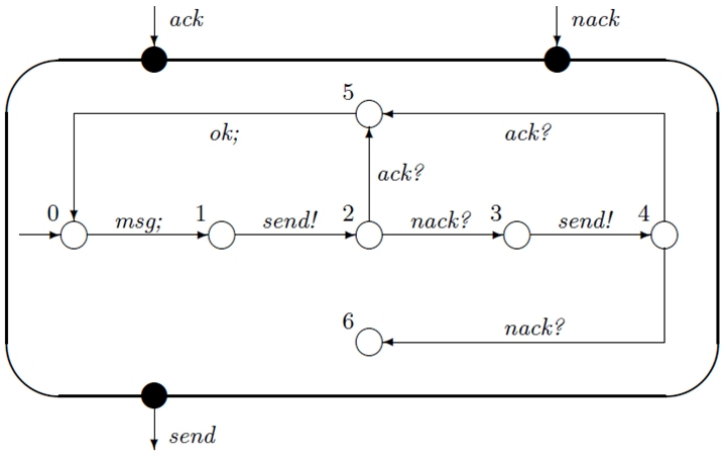

Fig. 8. the valid transition set in the IA form.

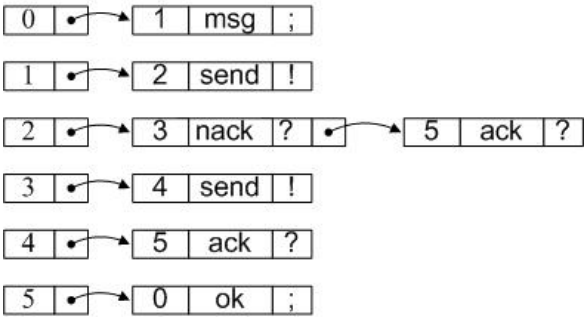

Fig. 9. the valid transition set in the adjacency-list form.

3, we create a state 0 in the comprehensive legal environment to correspond state 0 and state 1. Then we come to the second transition (1, send, 2). According to rule 1, we create a corresponding transition (0, send, 1) in the comprehensive legal environment. The third transition is (2, nack, 3). According to rule 1, we create a corresponding transition (1, nack, 2). Similarly, the transition (3, send, 4) corresponds to the transition (2, send, 3) in the comprehensive legal environment. When it comes to transition (4, ack, 5), we use rule 2 to create the corresponding transition (3, ack, 4). And so we can construct the comprehensive legal environment little by little.

Figure 10 shows the illustration of the comprehensive legal environment we construct and Figure 11 shows its adjacency list.

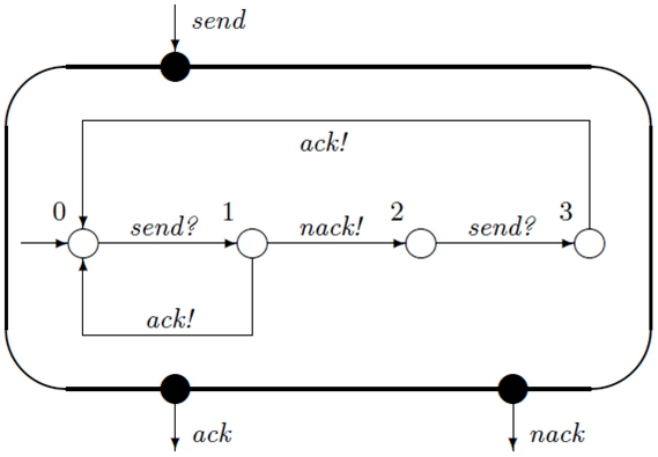

Fig. 10. the comprehensive legal environment in the IA form.

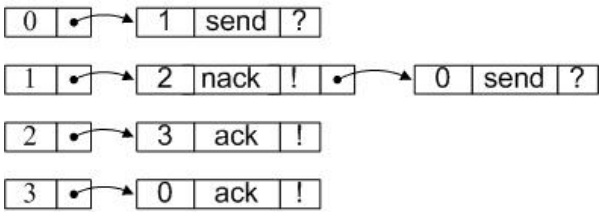

Fig. 11. the comprehensive legal environment in the adjacency-list form.

Figure12 shows the interface of the tool when we are running the case mentioned above.

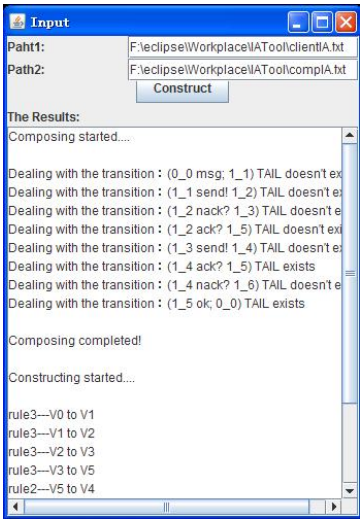

Fig. 12. the interface of the tool.

5 Analysis

In this section we will discuss the efficiency of the main algorithm.

Suppose there is one interface automaton R. The amount of states of R is Vr and the amount of transitions of R is Tr.

The time complexity of deriving the states of the valid state set is apparently O(Vr). And the time complexity of deleting related paths to get the valid transition is O(Tr). The time complexity of constructing the comprehensive legal environment is proportional to the square of Vr. However, in most situations, there will not be a transition between every two states. So the time complexity of constructing the comprehensive legal environment can be O(Tr).

So the time complexity of the whole algorithm is O(Vr+2Tr). Normally Vr is far less than Tr so the time complexity can be O(Tr).

6 References

[4] Yan Zhang, Jun Hu, Xiaofeng Yu, Tian Zhang, Xuandong Li, Guoliang Zheng. Deriving Available Behavior All Out from Incompatible Component Compositions. In: Proceedings of the 2nd International Workshop on Formal Aspects of Component Software (FACS 2005). Electronic Notes in Theoretical Computer Science, Vol. 160. Elsevier, 2006. 349-361

[5] de Alfaro, L. and T. A. Henzinger, Interface Automata, in: Proceedings of 9th Annual ACM Symposium on Foundations of Software Engineering (FSE 2001), ACM Press, New York (2001), 109-120.

[6] Heineman, G. T. and W. T. Councill, "Componet-Based Software Engineering: Putting the Pieces Together", Addison-Wesleg, Boston, 2001.