Deriving Available Behavior All Out from Incompatible Component Compositions

Yan Zhang\textsuperscript{1,2}, Jun Hu\textsuperscript{2}, Xiaofeng Yu\textsuperscript{2}, Tian Zhang\textsuperscript{2}, Xuandong Li\textsuperscript{2}, Guoliang Zheng\textsuperscript{2}

\textit{State Key Laboratory of Novel Software Technology}  
\textit{Department of Computer Science and Technology}  
\textit{Nanjing University, Nanjing, P. R. China}

Abstract

In component-based software development, an important problem is behavioral incompatibility in component compositions. That two components are behaviorally incompatible means that they cannot work together due to the mismatching order of exchanged messages between them. One of the approaches to solving this problem is to construct an environment for the incompatibility components and make it possible that they can work together in the environment. In practice, if two components, particularly commercial off-the-shelf (COTS) components, are behaviorally incompatible, we usually desire that most of the useful behavior can be preserved all out from their composition rather than discard them simply. In this paper, we use interface automata to model the behavior of components, and present an approach to deriving available behavior all out from incompatible component compositions. The main idea of the approach is to construct a \textit{comprehensive legal environment} (CLE) such that two incompatible components can work together and the behavior of their composition can be preserved as much as possible. The principle of CLEs is to obviate the incompatible points of behavior, i.e. the states at which an input action provided by one component is not accepted by the other, to be reached in the composition by selecting the appropriate input actions provided for the composition. We develop an algorithm to construct CLEs, and discuss the possible improvement for the algorithm.

Keywords: Component compositions, behavioral compatibility, interface automata, comprehensive legal environment

1 Introduction

The aim of components is large-scale reuse. Software systems can be developed by assembling and integrating existing software components, e.g., commercial off-the-shelf (COTS) components. By using components as reusable building blocks, we can rapidly and economically attain reliable, flexible, extensible and evolvable systems\cite{7}. Under component-based software development (CBSD), the producers...
can freely select, change and configure components to adapt requirements better. One of the major problems encountered in CBSD is component composition, i.e. how to make a number of components working together to perform a given task.

Component composition involves two different issues. One is compatibility, which includes interface compatibility and behavioral compatibility. The former means that two composed components match each other at their signatures (e.g., type matching about exchanged messages). The latter means 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. The other issue is synthesis, which means that the function of compositions is consistent with requirements of the user. Synthesis mainly involves semantics, functional equivalence, behavioral equivalence and so on.

The problem we concern in this paper is related to the behavioral compatibility. Suppose there are two components \( P \) and \( Q \). Their signatures match with each other, and the function of their composition is consistent with the requirements, but they aren’t behaviorally compatible, i.e. in the composition \( P \otimes Q \) there are several states, called illegal states, at which one component cannot accept the input action provided by the other component. One of the approaches to solving such behavioral incompatibility of \( P \) and \( Q \) is to construct an environment for them and make it possible that they can work together in the environment. That is, if we want to use \( P \otimes Q \), we need to find a component \( E \) such that the illegal states in \( (P \otimes Q) \otimes E \) cannot be entered forever. Furthermore, if \( E \) also satisfies that the most behavior of \( P \otimes Q \) is preserved in \( (P \otimes Q) \otimes E \), it is the result we want to get, which is called comprehensive legal environment (CLE). In this paper, we develop an algorithm to construct CLE for given components \( P \) and \( Q \).

In this paper, we use interface automata [5] to model the behavior of components, and the products of interface automata to represent the compositions of components. A primary reason for selecting interface automata as the modelling language is that the optimistic approach in interface automata theory suits to our problem to be solved. The so-called optimistic approach is that two components are compatible if there exists an environment (environment is also a component) that can make them work together without any error, which forms the base of the solution given in this paper. In comparison with other formal methods, such as traditional automaton, process algebra and temporal logic, which aren’t based on optimistic approach, interface automata can tackle the essential problem easily.

The remainder of this paper is organized as follows. Section 2 introduces the related concepts about interface automata. The algorithm to construct CLE is presented in section 3, and its possible improvement is discussed in section 4. Section 5 is about the related work, and the last section gives some conclusions.

2 Preliminary

In the section, most of concepts about interface automata refer to [5].

**Definition 2.1** An interface automaton \( P = \langle V_P, V_p^{\text{init}}, A_p^I, A_p^O, A_p^H, T_P \rangle \) is a 6-tuple, where
Definition 2.2

An execution fragment of interface automaton $P$ is a finite alternating sequence of states and actions $v_0 a_0 v_1 a_1 \cdots v_n$, where $(v_i, a_i, v_{i+1}) \in \mathcal{T}_P$, for all $0 \leq i < n$. Given two states $v, u \in V_P$, we say that $u$ is reachable from $v$ if there is an execution fragment with $v$ as the first state and $u$ as the last state. The state $u$ is reachable in $P$ if there is an initial state $v \in V_P^{\text{init}}$ such that $u$ is reachable from $v$.

Definition 2.3

Two interface automata $P$ and $Q$ are composable if

\[
\mathcal{A}_P^H \cap \mathcal{A}_Q = \emptyset \quad \mathcal{A}_P^I \cap \mathcal{A}_Q^I = \emptyset \\
\mathcal{A}_P^O \cap \mathcal{A}_Q^O = \emptyset \quad \mathcal{A}_Q^H \cap \mathcal{A}_P = \emptyset
\]
Let \( \text{shared}(P, Q) = \mathcal{A}_P \cap \mathcal{A}_Q = (\mathcal{A}_P^I \cap \mathcal{A}_Q^O) \cup (\mathcal{A}_P^O \cap \mathcal{A}_Q^I) \) be the set of shared actions of \( P \) and \( Q \).

**Definition 2.4** If two interface automata \( P \) and \( Q \) are composable, their product is the interface automaton defined by

\[
\begin{align*}
V_{P \otimes Q} &= V_P \times V_Q \\
V_{P \otimes Q}^{\text{init}} &= V_{P}^{\text{init}} \times V_{Q}^{\text{init}} \\
\mathcal{A}_{P \otimes Q}^I &= (\mathcal{A}_P^I \cup \mathcal{A}_Q^I) \setminus \text{shared}(P, Q) \\
\mathcal{A}_{P \otimes Q}^O &= (\mathcal{A}_P^O \cup \mathcal{A}_Q^O) \setminus \text{shared}(P, Q) \\
\mathcal{A}_{P \otimes Q}^H &= \mathcal{A}_P^H \cup \mathcal{A}_Q^H \cup \text{shared}(P, Q)
\end{align*}
\]

\[
\begin{align*}
T_{P \otimes Q} &= \{((v, u), a, (v', u')) | (v, a, v') \in T_P \land a \notin \text{shared}(P, Q) \land u \in V_Q \} \\
&\quad \cup \{((v, u), a, (v, u')) | (u, a, u') \in T_Q \land a \notin \text{shared}(P, Q) \land v \in V_P \} \\
&\quad \cup \{((v, u), a, (v', u')) | (v, a, v') \in T_P \land (u, a, u') \in T_Q \land a \in \text{shared}(P, Q) \}.
\end{align*}
\]

**Definition 2.5** Given two composable interface automata \( P \) and \( Q \), the set of illegal states of \( P \otimes Q \) is denoted by \( \text{Illegal}(P, Q) \subseteq V_P \times V_Q \) and is defined by

\[
\text{Illegal}(P, Q) = \{(v, u) \in V_P \times V_Q | \exists a \in \text{shared}(P, Q) .
\]

\[
((a \in \mathcal{A}_P^O(v) \land a \notin \mathcal{A}_Q^I(u)) \lor (a \in \mathcal{A}_Q^O(u) \land a \notin \mathcal{A}_P^I(v))) \}.
\]

Two composable interface automata \( P \) and \( Q \) are **behaviorally incompatible** if \( \text{Illegal}(P, Q) \neq \emptyset \).

**Definition 2.6** An interface automaton \( E \) is an environment for an interface automaton \( R \) if: (1) \( E \) is composable with \( R \), (2) \( E \) is not empty, and (3) \( \mathcal{A}_E^I = \mathcal{A}_R^O \). Given two composable interface automata \( P \) and \( Q \), a **legal environment** for them is an environment \( E \) for \( P \otimes Q \) such that no state in \( \text{Illegal}(P, Q) \times V_E \) is reachable in \((P \otimes Q) \otimes E\).

This definition of legal environment can be generalized as follows.

**Definition 2.7** Given an interface automaton \( P \) and a set of desired unreachable states \( V \subseteq V_P \setminus V_P^{\text{init}} \), a **legal environment** for \( P \) is an environment \( E \) for \( P \) such that no state in \( V \times V_E \) is reachable in \((P \otimes Q) \otimes E\).

### 3 Construction of Comprehensive Legal Environments

Given two behaviorally incompatible components \( \text{COMP1} \) and \( \text{COMP2} \), we want to find the third component \( \text{COMP3} \), such that \( \text{COMP1}, \text{COMP2} \) and \( \text{COMP3} \) are behaviorally compatible and the behavior of the composition of \( \text{COMP1} \) and \( \text{COMP2} \) should be preserved in the composition of \( \text{COMP1}, \text{COMP2} \) and \( \text{COMP3} \) as much as possible. If components \( \text{COMP1} \) and \( \text{COMP2} \) are specified as interface automata \( P \) and \( Q \), then \( \text{COMP3} \) can be found by constructing a legal environment \( E \) for \( P \otimes Q \), and the most steps in \( P \otimes Q \) are preserved in \((P \otimes Q) \otimes E\), i.e., the comprehensive legal environment for \( P \otimes Q \). So, interface automaton \( E \) equals to component \( \text{COMP3} \).
If regard \( P \otimes Q \) as an interface automaton \( R \) and \( \text{Illegal}(P, Q) \) as a subset of states in \( R \), according to Definition 2.7, then the above problem is translated to construct the comprehensive legal environment \( E \) for \( R \).

For simplicity, we make conventions for interface automata \( R \) and \( E \): (1) \( R \) and \( E \) are deterministic, (2) \( R \) and \( E \) are non-blocking, and (3) all states of \( R \) (resp. \( E \)) are reachable in \( R \) (resp. \( E \)). Without special statements, all interface automata referred hereinafter should conform to these conventions.

3.1 Foundation

Let \( \Gamma_R \) denote the set of all execution fragments in an interface automaton \( R \). For every state \( v_i \), \( 0 \leq i \leq n \) in an execution fragment \( \alpha = v_0a_0v_1a_1 \cdots v_n \in \Gamma_R \), we say \( v_i \) is on \( \alpha \) or \( v_i \) occurs on \( \alpha \), denoted by \( v_i \in \alpha \).

**Definition 3.1** Given an execution fragment \( \alpha = v_0a_0v_1a_1 \cdots v_n \in \Gamma_R \), if \( v_0 \in V_R^{\text{init}} \) and \( v_0 = v_n \), then \( \alpha \) is a circuit in \( R \). Specifically, if \( \alpha \) is a circuit in \( R \otimes E \), \( v_0 \in V_R^{\text{init}} \) and for any state \( v \in \alpha \), there isn’t any illegal state \( u \in \text{Illegal}(R, E) \) that is reachable from \( v \), then \( \alpha \) is a legal circuit in \( R \otimes E \). The set of all legal circuits in \( R \otimes E \) is denoted by \( \Sigma_{R \otimes E} \).

For any \( \beta, \beta' \in \Gamma_R \), if \( \beta' \) is a subsequence of \( \beta \), then we say that the execution fragment \( \beta' \) is on \( \beta \), denoted by \( \beta' \sqsubseteq \beta \). Specifically, if \( \beta' = vav' \) then we say that the step \( \tau = (v, a, v') \in T_R \) is on \( \beta \), denoted by \( \tau \sqsubseteq \beta \).

**Definition 3.2** The trace of an execution fragment \( \beta = v_0a_0v_1a_1 \cdots a_{n-1}v_n \in \Gamma_R \) is the subsequence of \( \beta \), which consists of all actions in \( \beta \). We write \( \text{trace}(\beta) = a_0a_1 \cdots a_{n-1} \). For any action \( a_i \), \( 0 \leq i \leq n-1 \) in \( \text{trace}(\beta) \), we say \( a_i \) is in \( \text{trace}(\beta) \), denoted by \( a_i \in \text{trace}(\beta) \).

**Definition 3.3** Given an execution fragment \( \beta \in \Gamma_{R \otimes E} \) and \( \text{trace}(\beta) = a_0a_1 \cdots a_{n-1} \), the projection of the trace of \( \beta \) on \( R \) is a subsequence of \( \text{trace}(\beta) \), which is obtained by deleting all actions \( a_i \in A_E \setminus \text{shared}(R, E) \), \( 0 \leq i \leq n-1 \) in \( \text{trace}(\beta) \). The projection of the trace of \( \beta \) on \( R \) is denoted by \( \pi_R(\text{trace}(\beta)) \).

There are an execution fragment \( \beta = v_1a_1v_2a_2 \cdots v_n \) of interface automaton \( R \) and a legal environment \( E \) of \( R \). \( \beta \) is covered by legal circuits of \( R \otimes E \) if there exists an execution fragment \( \gamma = (v_1, u_1)a_1(v_2, u_2)a_2 \cdots (v_n, u_n) \) in \( R \otimes E \) that is on a legal circuit of \( R \otimes E \) and \( \pi_R(\text{trace}(\gamma)) = \text{trace}(\beta) \). For short, \( \beta \) is covered by \( \Sigma_{R \otimes E} \). Specifically, if \( \beta = vav' \) then step \( \tau = (v, a, v') \in T_R \) is covered by \( \Sigma_{R \otimes E} \).

**Definition 3.4** Two legal environments \( E \) and \( E' \) of interface automaton \( R \) are equivalent if they satisfy the following conditions

\[
\forall \alpha \in \Sigma_{R \otimes E}, \exists \beta \in \Sigma_{R \otimes E'}, \pi_R(\text{trace}(\alpha)) = \pi_R(\text{trace}(\beta)) \quad \text{and} \quad \forall \beta \in \Sigma_{R \otimes E'}, \exists \alpha \in \Sigma_{R \otimes E}, \pi_R(\text{trace}(\alpha)) = \pi_R(\text{trace}(\beta)).
\]

This is written by \( E \simeq E' \).

**Definition 3.5** A legal environment \( E \) of interface automaton \( R \) is comprehensive legal environment of \( R \), if and only if there doesn’t exist any execution fragment in
There are

Theorem 3.8 must exist a legal environment $\tau$ for any interface automaton $\mathcal{A}$.

For any interface automaton $\mathcal{A}$ and $\tau$, there must be a loop in $\mathcal{A}$.

Theorem 3.9 demonstrates that if there are two execution fragments in the

Theorem 3.10 demonstrates that if there are two execution fragments in the interface automaton $\mathcal{A}$, and their forms are similar to $\beta_1$ and $\beta_2$ respectively, then
the execution fragment, whose form is similar to $\beta_2$, can’t be preserved in the composition of $R$ and $R$’s any legal environment.

Suppose that interface automaton $R$ has a legal environment $E$ and there are step $\tau_R = (v, a, v')$ in $R$ and step $\tau_E = (u, a, u')$ in $E$. If $\tau_{R \otimes E} = ((v, u), a, (v', u'))$ is on certain legal circuit of $R \otimes E$, then $\tau_E$ is called the corresponding step of $\tau_R$ in the legal environment $E$ of $R$, and states $u$, $u'$ are called the corresponding states of $v$, $v'$ respectively. Specifically, if $\text{label}(\tau_R) \notin \text{shared}(R, E)$, then there isn’t a corresponding step of $\tau_R$ in $E$, but there are corresponding states of $\text{head}(\tau_R)$ and tail($\tau_R$).

**Theorem 3.11** The interface automaton $R$ has a legal environment $E$ that holds $A_E = \text{shared}(R, E)$. There is a circuit $\alpha = v_0 a_0 v_1 a_1 \cdots v_0$ in $R$ and $\alpha$ is covered by certain legal circuit of $R \otimes E$. For a step $\tau_R = (v_i, a_i, v_{i+1}) \subseteq \alpha$, write the execution fragment on $\alpha$ succeeding $\tau_R$ as $\beta$, i.e., $\beta = v_{i+1} a_{i+1} \cdots v_0$. If $\forall a \in \text{trace}(\beta). a \notin \text{shared}(R, E)$, then for $\tau_E = (u_i, a_i, u_{i+1})$ that is the corresponding step of $\tau_R$ in $E$, $u_{i+1}$ must be the initial state of $E$.

Theorem 3.11 demonstrates that if there are a circuit in the interface automaton $R$ and a step $\tau_R$ on the circuit, and they satisfy the conditions in Theorem 3.11, then in the legal environment $E$ of $R$ the corresponding step of $\tau_R$ must point to the initial state of $E$, otherwise there doesn’t exist any legal circuit in $R \otimes E$ that can cover the circuit in $R$.

Because of omitting all internal actions in the legal environment according to Theorem 3.8, a complex case maybe appear in the process of constructing legal environment. Figure 2 shows this case. Suppose that execution fragment $v_1 a v_3 c v_4$ and $v_2 b v_3 c v_4$ are on different circuits in an interface automaton $R$ and we have constructed corresponding steps for $(v_1, a, v_3)$ and $(v_3, c, v_4)$ in the legal environment $E$ of $R$. Since $b$ is an internal action of $R$ and there may be no internal action in $E$ according to Theorem 3.8, in this case, the corresponding state of $v_2$ in $E$ maybe differ with that of $v_3$ in $E$. Hence, execution fragment $v_2 b v_3 c v_4$ couldn’t be covered by any legal circuit of $R \otimes E$, but it could be covered in fact if the corresponding state of $v_2$ is identical with that of $v_3$ in $E$. There are many methods to solve the problem. If make a convention that for one step in $R$ there is unique corresponding step of its in $E$, then we can obtain a theorem as follows.

**Theorem 3.12** The interface automaton $R$ has a legal environment $E$. There are execution fragments $\beta_1 = v_i a_i v_j a_j v_k$ and $\beta_2 = v_i a_i v_j a_j v_k$ in $R$, where $i \neq j \neq k \neq t$, $a_i \neq a_j \neq a_k \neq a_t$, $a_t \notin \text{shared}(R, E)$. $\beta_1$ and $\beta_2$ are covered by $\Sigma_{R \otimes E}$ and the corresponding state of $v_j$ is state $u_j$ in $E$. If there exists unique corresponding step of $(v_j, a_j, v_k) \in T_R$ in $E$, then the corresponding state of $v_t$ must be $u_j$ too.
Given an interface automaton $R$ and the set of desired unreachable states $V \subseteq V_R \setminus V_R^{\text{init}}$, if there exists a legal environment $E$ of $R$, then $E$ can obviate to reach states in $V \times V_E$ only by selecting appropriate input actions provided for $R$, because $E$ must accept all output actions from $R$. Thus, any state in $R$ from which can enter states in $V \times V_E$ must be unreachable too.

According to [5], we introduce the operator $\text{OHpre}_R : 2^{V_R} \rightarrow 2^{V_R}$, which is defined for all $U \subseteq V_R$ by

$$\text{OHpre}_R(U) = \{ v \in V_R | \exists (v, a, v') \in T^O_R \cup T^H_R . v' \in U \}.$$ 

Then, we define the valid state set $V_{\text{valid}}_R$ of $R$ as $V_{\text{valid}}_R = V_R \setminus V^*$, where $V^* = \nu X . X \cup \text{OHpre}_R(X)$, $X_0 = V$ and $\nu$ is the greatest fixpoint operator. If $V_R^{\text{init}} \subseteq V_{\text{valid}}_R$ then we can define the valid step set $T_{\text{valid}}_R$ of $R$ as $T_{\text{valid}}_R = T_R \cap (V_{\text{valid}}_R \times A_R \times V_{\text{valid}}_R)$.

**Theorem 3.13** If interface automaton $R$ have legal environments, then for every legal environment $E$ of $R$, circuits in $R$ that can be covered by $\Sigma_R \otimes E$ all belong to the set of circuits in $R$ that are composed of steps in valid step set of $R$.

Theorem 3.13 demonstrates that for every circuit in the interface automaton $R$, if it can be covered by a legal circuit in the product of $R$ and $R$’s legal environment, then the circuit must be composed of steps in valid step set of $R$, i.e., steps in $T_{\text{valid}}_R$.

**Theorem 3.14** For any interface automaton $R$, if there are legal environments of $R$, then there must exist a comprehensive legal environment of $R$.

### 3.2 Constructive Algorithm

**Methodology**

As mentioned previously, the legal environment $E$ of an interface automaton $R$ can obviate to enter states in $V \times V_E$, where $V$ is the set of undesired reachable states in $R$, only by selecting appropriate input actions provided for $R$. Hence, firstly compute the valid state set $V_{\text{valid}}_R$ of $R$. For any input step $\tau_R$ in $R$, if some states in $V_R \setminus V_{\text{valid}}_R$ can be reachable from some states in $V_{\text{valid}}$ by taking the $\tau_R$, the corresponding step of $\tau_R$ isn’t constructed in $E$ uniformly. Secondly, according to Theorem 3.13 and 3.10, we can obtain the set that contains the most circuits in $R$ and these circuits can be covered by legal circuits in the product of $R$ and $R$’s legal environment. If the set is empty then there doesn’t exist CLE for $R$. Otherwise, select a circuit from the set arbitrarily and from the initial state of $R$ make depth-first traversal along steps on the circuit. For every traversed step of $R$, according to rules of constructing corresponding steps (see, Fig. 3), construct its corresponding step in $E$. When the traversal of one circuit ends, select another circuit from the set and repeat the process until all circuits in the set have been traversed.
Rule 1. If $\tau_R$ is a loop on $\text{head}(\tau_R)$, then $\tau_E$ is a loop on $\text{head}(\tau_E)$; if $\tau_R$ isn’t a loop on $\text{head}(\tau_R)$, then $\tau_E$ isn’t a loop on $\text{head}(\tau_E)$. Exception is that $\tau_E$ must be a loop by Rule 2. $\text{head}(\tau_E)$ is the corresponding state of $\text{head}(\tau_R)$.

Rule 2. If there is circuit $\alpha = v_0 \cdots v a v' a' \cdots v_0$ in $R$, then write execution fragment $v' a' \cdots v_0 \subseteq \alpha$ as $\beta$. If $\forall e \in \text{trace}(\beta), e \notin \text{shared}(R, E)$, then there must be tail($\tau_E$) is $v_0$, where $\tau_R = (v, a, v')$ and $u_0$ is the initial state of $E$.

Rule 3. If there is step $\tau'_R$ in $R$, tail($\tau'_R$) = tail($\tau_R$) and there has been an corresponding step of $\tau'_E$ in $E$, say $\tau'_E$, then let tail($\tau_E$) = tail($\tau'_E$). Otherwise, let tail($\tau_E$) be a new state not in $V_E$.

Rule 4. Along the circuit on which $\tau_R$ is, scan the step succeeding $\tau_R$, say $\tau_R'$:

(i) if label($\tau_R'$) $\notin \text{shared}(R, E)$ and there has been a corresponding state of tail($\tau'_R$) in $E$, say $u$, then let tail($\tau_E$) = $u$;

(ii) if label($\tau'_R$) $\notin \text{shared}(R, E)$ and there hasn’t been a corresponding state of tail($\tau'_R$) in $E$, then scan the step succeeding $\tau'_R$ along the same circuit, say $\tau''_R$ and take $\tau''_R$ as $\tau''_R$ to apply Rule 4 again;

(iii) if label($\tau'_R$) $\in \text{shared}(R, E)$, then construct $\tau_E$ according to Rule 1 to 3.

Fig. 3. The rules of constructing corresponding steps in $E$ for steps in $R$, where $E$ is the constructed CLE of a given interface automaton $R$, $\tau_R$ is a step in $R$, $\tau_E$ is the corresponding step of $\tau_R$ in $E$ and set $V_E$ records all states in $E$ at every time during the constructing process.

Algorithm

Make the convention of $V_R^{\text{init}} = \{v_0\}$, $A_E^H = \emptyset$ and $A_R^I = A_R^O$. The constructive algorithm of CLE $E$ for a given interface automaton $R$ is shown in Algorithm 1, where $V$ is the set of all undesired reachable states in $R$.

Algorithm 1 Input: Interface automaton $R$ and $V \subseteq V_R \setminus V_R^{\text{init}}$.
Output: Comprehensive legal environment $E$ of $R$.

Step 1: Compute the valid state set of $R$, $V^{\text{valid}}_R = V_R \setminus V^*$, where $V^* = vX. X \cup O\text{Hpre}_R(X)$, $X = V$, and the valid step set of $R$, $T^{\text{valid}}_R = T_R \cap (V^{\text{valid}}_R \times A_R \times V^{\text{valid}}_R)$.

Step 2: According to Theorem 3.10, find all execution fragments in $R$ that cannot be covered by any legal circuit in the product of $R$ and any $R$’s legal environment, and use $T^*$ to denote the set of these execution fragments.

Step 3: In all circuits that are composed of steps in $T^{\text{valid}}_R$, delete all elements in $T^*$. Denote the set of remainder circuits in $R$ after above modification as $\Sigma$.

Step 4: If $\Sigma = \emptyset$ then CLE $E$ of $R$ doesn’t exist and the algorithm terminates, else initialize CLE $E$ of $R$, i.e. let $V_E = \{u_0\}$, and enter Step 5.

Step 5: Select a circuit $\alpha$ from $\Sigma$ arbitrarily and from $v_0$ traverse steps on $\alpha$ by depth-first. For every traversed step $\tau_R$, according to the rules shown in Fig. 3, construct $\tau_R$’s corresponding step $\tau_E$ in $E$ and add tail($\tau_E$) to $V_E$. When all steps on $\alpha$ have been traversed, mark $\alpha$ as processed circuit and select another unprocessed circuit in $\Sigma$ and repeat Step 5.

Step 6: If all circuits in $\Sigma$ are processed, then the algorithm terminates and CLE $E$ of $R$ is returned.

The rules of constructing corresponding steps in the CLE $E$ of the given interface automaton $R$ for steps in $R$ are shown in Fig. 3.
Fig. 4. The CLE $E$ for $R$ shown in Fig. 1 constructed by Algorithm 1. Because state 6 in $R$ is desired unreachable, the action $fail$ cannot be output and it is omitted in $E$.

Fig. 5. The composition $R \otimes E$

Analysis

Because Algorithm 1 uses the depth-first traversal and the set of states of $R$ is finite, the algorithm can terminate confidently. The kernel of Algorithm 1 is rules shown in Fig. 3. Theorem 3.9, 3.11 and 3.12 guarantee the correctness of Rule 1, 2 and 4 respectively. Theorem 3.8, 3.10, 3.13 and 3.14 guarantee the correctness of Step 1 to 4 and 6 in Algorithm 1. In conclusion, the constructive algorithm is correct.

The complexity of Algorithm 1 is proportional to the number of states in the valid state set of the interface automaton $R$.

Example 3.15 The CLE $E$ (see, Fig. 4) of the interface automaton $R$ (see, Fig. 1) is constructed by Algorithm 1, where the set of desired unreachable states is $V = \{6\}$. The composition $R \otimes E$ is shown in Fig. 5.

4 Discussion about Improvement of the Constructive Algorithm

The composition of interface automaton $R$ and its CLE $E$ constructed by Algorithm 1 is closed. In practice, we desire that the composition is open, and then it can compose with other interface automata again. Accordingly, we want to make improvement for Algorithm 1 on the following aspect:

Introduce actions not in shared($R, E$) into $E$ constructed by Algorithm 1 and assure that the result is still a legal environment of $R$.

Define $\text{Th}_R(v) = \{ \tau \mid \tau \in T_R \land \text{head}(\tau) = v \land v \in V_R \}$ and $\text{Tt}_R(v) = \{ \tau \mid \tau \in T_R \land \text{tail}(\tau) = v \land v \in V_R \}$, which are the set of all steps that depart from and point to the state $v$ in the interface automaton $R$ respectively. Define
Each of them is specified with its applicable situation and mechanism. But their needed of consumers. In their paper, four types of smart connectors are given and addressed the problem in literatures.

Further, it is also important that the protocol compatible is significant to enable collaboration of two components when they are incompatible to work together. Therefore, appropriate connectors or adaptors can be automatically generated for components with incompatible behavior.

For every step that departs from \( v_i \), \( \text{Th} \text{change}_{R}(v_i, v_j) \) copies a same step that departs from \( v_j \). For every step that points to \( v_i \), \( \text{Th} \text{change}_{R}(v_i, v_j) \) copies a same step that points to \( v_j \).

**Proposition 4.1** \( E = \langle V_{E}, V_{E}^{\text{init}}, A_{E}^{I}, A_{E}^{O}, A_{E}^{H}, T_{E} \rangle \) is a legal environment of interface automaton \( R \). \( w_{ij} = (v_{i}, u_{j}) \in V_{R \otimes E} \) and \( w_{ij} \) is on a legal circuit of \( R \otimes E \). \( A_{R}(v_i) \subseteq A_{R}^{I} \cap A_{E}^{O} \). \( E' = \langle V_{E}', V_{E}'^{\text{init}}, A_{E}'^{I}, A_{E}'^{O}, A_{E}'^{H}, T_{E}' \rangle \) and \( R \) are composable, where \( V_{E'} \setminus V_{E} = \{ u_{j}' \} \), \( V_{E}'^{\text{init}} = V_{E}^{\text{init}} \), \( A_{E}' \setminus A_{E} = \{ a \} \), \( a \notin \text{shared}(R, E') \), \( T_{E}' = (T_{E} \setminus \text{Th} \text{ead}_{E}(u_j)) \cup \text{Th} \text{change}_{E}(u_j, u_j') \cup \{ (u_j, a, u_j) \} \), then \( E' \simeq E \).

**Proposition 4.2** \( E = \langle V_{E}, V_{E}^{\text{init}}, A_{E}^{I}, A_{E}^{O}, A_{E}^{H}, T_{E} \rangle \) is a legal environment of interface automaton \( R \). \( w_{ij} = (v_{i}, u_{j}) \in V_{R \otimes E} \) and \( w_{ij} \) is on a legal circuit of \( R \otimes E \). \( A_{R}(v_i) \cap (A_{R}^{O} \cap A_{E}^{I}) = \emptyset \). \( E' = \langle V_{E}', V_{E}'^{\text{init}}, A_{E}'^{I}, A_{E}'^{O}, A_{E}'^{H}, T_{E}' \rangle \) and \( R \) are composable, where \( V_{E'} \setminus V_{E} = \{ u_{j}' \} \), \( V_{E}'^{\text{init}} = V_{E}^{\text{init}} \), \( A_{E}' \setminus A_{E} = \{ a \} \), \( a \notin \text{shared}(R, E') \), \( T_{E}' = (T_{E} \setminus \text{T} \text{tail}_{E}(u_j)) \cup \text{Th} \text{change}_{E}(u_j, u_j') \cup \{ (u_j, a, u_j) \} \). If \( E' \) is a legal environment of \( R \), then \( E' \simeq E \).

Proposition 4.1 and 4.2 indicate that we can add some steps whose labels don’t belong to \( \text{shared}(R, E) \) on some states of the legal environment \( E \) of the interface automaton \( R \). If let \( E' \) denote the new legal environment, then Proposition 4.1 guarantees that \( E' \) must exist and can be constructed by the method in the proposition, and Proposition 4.2 doesn’t guarantee that \( E' \) must exist, but if \( E' \) exists then it can be constructed by the method in the proposition.

## 5 Related Work

The primary concern of individual component is functionality, but the component based software systems primarily focus on component interactions. Particularly, it is significant to enable collaboration of two components when they are functionally compatible but they aren’t protocol compatible [10]. The connectors [1] or adaptors, which is similar to the legal environment, can be used to make behaviorally incompatible components to work together. Further, it is also important that the appropriate connectors or adaptors can be automatically generated for components with incompatible behavior. There have been various pieces of research about this problem in literatures.

Min et al. [8] use the connectors to solve the partial matching problem, i.e. the candid component couldn’t completely satisfy the functionality and interface needed of consumers. In their paper, four types of smart connectors are given and each of them is specified with its applicable situation and mechanism. But their
research work doesn’t involve how to automatically generate these connectors.

In [10], Yellin and Strom define software adaptor that allows the composition of protocol incompatible components and give the algorithm that can automatically generate adaptor for protocol incompatible components according to a high-level description, called an interface mapping. Their approach is based on finite state machines in essence.

Similarly, Bracciali et al. [4] also describe an approach to solve mismatching behaviors between components by adaptors. In their approach, adaptor derivation can automatically generate a concrete adaptor from adaptor specifications that specify interoperation between two components. They use $\pi$-calculus to model behavior of components and give the algorithm about adaptor derivation.

Formalisms of [10] and [4] are all based on the pessimistic approach [5], therefore both of them need the assistance of a specification of adaptors to synthesize the adaptor. Our proposal uses interface automata to model component behavior, which integrate assumptions about the environment into the behavior model of components, thus legal environment can be constructed without any specification of environment. However, approaches of [10] and [4] can solve interface incompatibility as well as behavioral incompatibility. Moreover, [10] takes into account non-functional behavior.

In [6], authors adopt the optimistic approach to some extent for model checking of software component. They give algorithms that can automatically generate an assumption that characterizes exactly those environments in which the component satisfies its required property. In our approach, constructed CLE for components contains not only the assumption about the environment but also concrete behavior about the environment.

In discrete event systems [9], the synthesis problem [3,2] is most pertinent to our research. In the theory of control of discrete event systems, process is a deterministic non-complete finite state automaton over an alphabet of events. The controller is a process that must react to any uncontrollable event and cannot detect the occurrence of an unobservable event. The synthesis problem is to construct a controller $R$ for process $P$, such that all behavior of the supervised system composed by $P$ and $R$ are admissible. Constructing controller amounts to finding winning strategies in parity games. Bernet et al. [3] research on the permissive strategy that contains most behavior of system than other winning strategies do. By an analogy, it is interesting to notice that the legal environment corresponds to the controller in discrete event systems and the constructed comprehensive legal environment in this paper corresponds to the controller under the permissive strategy. In comparison with [3], we tackle the problem from behavior perspective rather than from game perspective. The complexity of our algorithm is better than theirs.

6 Conclusion

In this paper, we use interface automata to model the behavior of components, and present an approach to deriving available behavior all out from incompatible
component compositions. The main idea of the approach is to construct a comprehensive legal environment (CLE) such that two incompatible components can work together and the behavior of their composition can be preserved as much as possible. The principle of CLEs is to obviate the incompatible points of behavior, i.e. the states at which an input action provided by one component is not accepted by the other, to be reached in the composition by selecting the appropriate input actions provided for the composition. We develop an algorithm to construct CLEs, and discuss the possible improvement for the algorithm.

In the future, we are to extend our work for extracting the desired behavior exactly from the composition of components in terms of the requirements.

References


