Component Design and Adaptation Based on Behavioral Contracts

Samir Chouali, Sebti Mouelhi, Hassan Mountassir

Component Design and Adaptation Based on Behavioral Contracts

Samir Chouali[1], Sebti Mouelhi[2], and Hassan Mountassir[1]

[1] Univ. Bourgogne Franche-Comté, FEMTO-ST Institute/CNRS, Besançon, France
schouali@femto-st.fr, hmountas@femto-st.fr
[2] ECE Paris - Graduate School of Engineering, Paris, France
sebti.mouelhi@ece.fr

Abstract. In this paper, our objective is to propose an adaptation approach to generate a component adaptor that ensures a correct interaction between mismatched components. Compared to the related works on component adaptation, the originality of our proposition relies on two main contributions. In the first, we design component behavioral contracts in order to generate component adaptor. So, we propose to specify component interfaces as behavioral contracts, to enrich the exhibited informations in component interfaces. Our behavioral contracts express all component facets: their action signatures, their actions semantics, and their protocol. We consider that these informations are important when generating component adaptors. In the second contribution, we propose to specify component behavioral contracts with the formalism based on interface automata that we enrich to specify the semantics of component actions. So, our adaptation approach is also an extension of the interface automata approach to handle the problem of component adaptation.

Keywords: Components · behavioral contracts · adaptation.

1 Introduction

The development of component-based systems is principally based on component reusability which allows the use of components in diverse environments without affecting their codes. However, in many cases, reusability is constrained with mismatches that may occur between components and their new environments during their interaction. The mismatches are caused by components that do not match perfectly the requirements of their environment. In this case, component adaptation should be performed in order to generate software entities, called *adaptors*, capable of enabling a correct interaction between components when mismatches occur.

In this paper, we focus on adapting components whose interfaces are described with behavioral contracts, which exhibit all component facets at the levels of action signatures (signatures of component operations), component protocols (scheduling of operation calls), and action semantics (semantic of component operations). We believe that consideration of all these informations in component interfaces lead to generate suitable and reliable adaptors. To specify formally component contracts, we propose to exploit the interface automata formalism [1] that we enrich by the semantic of component actions, because interface automata express only the scheduling of components actions without their semantics. So, we annotate the actions in interface automata by pre and post-conditions expressed on their parameters. This new formalism led us to adapt the compatibility verification approach, based on interface automata, to handle with the semantic of actions, because the adaptor generation relies on the verification of component compatibility.

Previously, we treated only adaptation at the protocol level [6]. Our purpose was to generate automatically an adaptor (interface automaton in-the-middle) for exactly two component interface automata according to a mapping that establishes a number of rules relating their mismatched input and output actions. In this paper, the main contribution relies on proposing a methodological approach to treat the problem of component adaptation at signature, semantic, and protocol levels, by exploiting component behavioral contracts. We show how to cooperate between the adaptation at the protocol level, and the semantic adaptation to generate a suitable adaptor for components specified with enriched interface automata that specify component contracts.

The paper is organized as follows. In Section 2, we present the formalization of component behavioral contracts with the interface automata, enriched with the semantics of component actions. In Section 3, we show how to verify the compatibility between components specified with behavioral contracts. When the compatibility does not hold between components, we present in Section 4 the specification of the mapping rules between the mismatched components that we exploit to generate adaptors. Section 5, is dedicated to present our proposition to adapt components at signature, semantic, and protocol levels. Finally, we discuss the related work to our approach in Section 6 and conclude the paper in Section 7.

2 Component Behavioral Contracts

Interface automata (IAs) have been defined by L. Alfaro and T. Henzinger [1], to model the behavior of software component. Every component is described with an interface, which is specified with one interface automaton. This latter describes the scheduling of input, output, and hidden component actions, such that, input actions are used to model methods that can be called, and the end of receiving messages from communication channels, as well as the return values from such calls. Output actions are used to model method calls, message transmissions via communication channels, and exceptions that occur during the methods executions. Local operations are called hidden actions. The alphabet of an interface automaton consists of the names of actions annotated by "?" for input actions, by "!" for output actions, and by ";" for hidden actions. In the interface automata approach, the verification of the compatibility between two component is based on the composition of their interface automata, which is achieved by synchronizing their shared input and output actions. The compatibility holds between two interface automata where there is an environment (third component) which prevents the reachability of *illegal states* (states where the synchronization between the shared actions is not achieved) in their composition. This approach is considered optimistic because the existence of illegal states in the composition is not sufficient to decide on the incompatibility between components. The composition approach of the other automata-based formalisms describing the interface protocols of components are considered pessimistic.

In this paper, we propose to specify component behavioral contracts with interface automata formalism, enriched with the explicit description of the semantics of each action. These contracts specify component behaviors by showing the scheduling of the actions calls, and the interface automata formalism is suitable to specify component behaviors. However our behavioral contracts should express also the semantics of component actions, with pre and post conditions that should be satisfied by the environment in order to call or to provide component actions. And interface automata are not enough expressive to specify the semantics of component actions, therefore we propose to enrich this formalism to cope with action semantics.

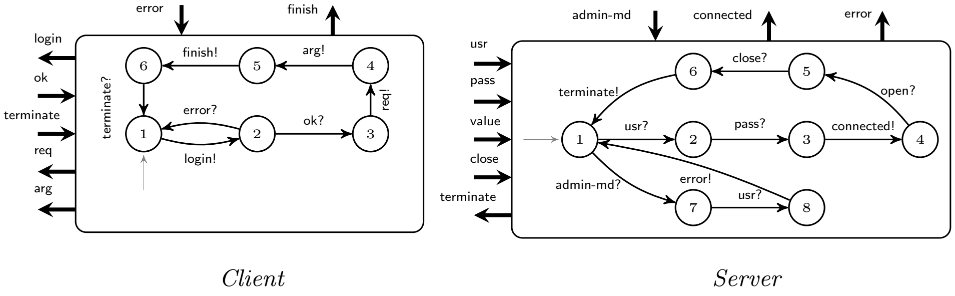
In our proposal, we consider that the signature of an input (resp. output) component action a is of the form $a(i_1,...,i_n) \to (o)$. The set $P_a^{\mathrm{i}} = \{i_1, ..., i_n\}$ represents the set of input parameters of a. The set of output parameters P_a^{o} is defined by the singleton $\{o\}$ (we assume that an action has at most a unique return value). The set of all parameters of an action a is denoted by P_a. The absence of input or output parameters is denoted by (). For a parameter p, we define a domain D_p which is a set of values that p can take. The semantics of actions is represented by the pre and post-conditions defined on action parameters. We express these conditions as formulas of the first-order logic. Given a set of variables V, we denote by $\mathit{Preds}(V)$, the set of first-order logic predicates whose free variables belong to V.

Definition 1 (Interface Automaton for a behavioral contract) *Let B be a behavioral contract associated to a component interface.* An *interface automaton* to specify B is a tuple $A = \langle \mathrm{S}_A, \mathrm{i}_A, \Sigma_A^{\mathrm{I}}, \Sigma_A^{\mathrm{O}}, \Sigma_A^{\mathrm{H}}, \delta_A, \Psi_A \rangle$ such that:

- S_A is a finite set of states. A is called *empty* iff $S_A = \emptyset$;
- $i_A \in S_A$ is the initial state;
- Σ_A^{I}, Σ_A^{O}, and Σ_A^{H} are respectively the sets of names of input, output, and hidden actions. Let $\Sigma_A = \Sigma_A^{\mathrm{I}} \cup \Sigma_A^{\mathrm{O}} \cup \Sigma_A^{\mathrm{H}}$;
- $\delta_A \subseteq S_A \times \Sigma_A \times S_A$ is the set of transitions betweens states;
- Ψ_A is a function, $\Psi_A : \Sigma_A \mapsto \mathit{Preds}(P_a^{\mathrm{i}}) \times \mathit{Preds}(P_a^{\mathrm{i}} \cup P_a^{\mathrm{o}})$, that associates, for each action a $\in \Sigma_A$, a tuple $\langle \mathrm{Pre}_{\Psi_A(a)}, \mathrm{Post}_{\Psi_A(a)} \rangle$ that represents the pre and post conditions of component actions.

We require that interface automata are deterministic, *i.e.* for all $(s, a, s_1) \in \delta_A$ and $(s, a, s_2) \in \delta_A$, we have $s_1 = s_2$.

The set Σ_A^{ext} of *external* actions of interface automaton A is defined by the union $\Sigma_A^{\mathrm{I}} \cup \Sigma_A^{\mathrm{O}}$. The set Σ_A^{loc} of *locally controlled* actions of A is defined by the union $\Sigma_A^{\mathrm{O}} \cup \Sigma_A^{\mathrm{H}}$. We define by $\Sigma_A^{\mathrm{I}}(s)$, $\Sigma_A^{\mathrm{O}}(s)$, $\Sigma_A^{\mathrm{H}}(s)$, $\Sigma_A^{\mathrm{ext}}(s)$, and $\Sigma_A^{\mathrm{loc}}(s)$ respectively the input, output, hidden, external and locally controlled actions enabled from s. The set $\Sigma_A(s)$ includes all the enabled actions from s.



Client *Server*

Fig. 1. A variant of a client/server system

Example 1. Let us consider the two composable interface automata *Client* and *Server*, that specify component behavioral contracts, shown in Figure 1. After authentication, *Client* sends a request *req*! to open a file in read-only or write mode. After that, it sends an action *arg*! containing the name of a file to be open. *Server* receives the two actions by executing an action *open*? that open the file in readonly or write mode. After using the file, *Client* sends a signal *finish*! indicating to *Server* that the file is ready to be closed (action *close*?). Finally, *Server* sends a signal *terminate*! to terminate the session. The action *admin-md*? is a super signal received from the administrator of the system to open a super user session. When a client username is received by the server after receiving the *admin-md*! signal from an administrator, then an error is detected. For example, the signatures and the semantics of the action *login* in *Client* and *usr* in *Server* are defined as follows.

Signatures: $login(uname, passwd, lu, lp) \rightarrow (exist)$, $usr(username, lengthu) \rightarrow ()$.

The semantics of the action *login* is defined as:
$Pre_{\Psi_{Client}(login)} \equiv 1 < \text{lu} \leq 20 \wedge 8 \leq \text{lp} \leq 10$,
$Post_{\Psi_{Client}(login)} \equiv \text{exist} = 1 \vee \text{exist} = 0$

The semantics of the action *usr* is defined as:
$Pre_{\Psi_{Server}(usr)} \equiv 1 < \text{lengthu} \leq 30$, $Post_{\Psi_{Server}(usr)} \equiv \text{true}$

3 Component Compatibility Based on Behavioral Contracts

In this section we show how to verify the compatibility between two components specified with their behavioral contracts. Our proposition relies on the extension of the interface automata approach to cope with the semantics of component actions expressed with their pre and post condition on their parameters. To verify the compatibility between two components that are specified with two interface automata A_1 and A_2, we have first to verify their composability and then compute their composition by their synchronized product.

Before defining the composition between, A_1 and A_2, we present in the following the conditions that should be respected by both automata, that specify component behavioral contracts, in order to authorize their composition.

The Composability conditions: Two interface automata A_1 and A_2 associated to two behavioral contracts are *composable* if :

- The condition on the non shared input and output actions is satisfied:
$$\Sigma^{\mathrm{I}}_{A_1} \cap \Sigma^{\mathrm{I}}_{A_2} = \Sigma^{\mathrm{O}}_{A_1} \cap \Sigma^{\mathrm{O}}_{A_2} = \Sigma^{\mathrm{H}}_{A_1} \cap \Sigma_{A_2} = \Sigma^{\mathrm{H}}_{A_2} \cap \Sigma_{A_1} = \emptyset.$$
- The condition on the shared actions is satisfied: *Shared*$(A_1,A_2) = (\Sigma^{\mathrm{I}}_{A_1} \cap \Sigma^{\mathrm{O}}_{A_2}) \cup (\Sigma^{\mathrm{I}}_{A_2} \cap \Sigma^{\mathrm{O}}_{A_1})$ is the set of shared input and output actions of A_1 and A_2. For each action $a \in$ *Shared*(A_1,A_2) such that its signature is given by $a(i_1,...,i_n) \rightarrow (o)$ in A_1 and by $a(i'_1,...,i'_n) \rightarrow (o')$ in A_2 then, $D_{i_k} \subseteq D_{i'_k}$ for $1 \leq k \leq n$ and $D_o \subseteq D_{o'}$ in the case where $a(i_1,...,i_n) \rightarrow (o) \in \Sigma^{\mathrm{O}}_{A_1}$ and $a(i'_1,...,i'_n) \rightarrow (o') \in \Sigma^{\mathrm{I}}_{A_2}$. Otherwise, $D_{i_k} \supseteq D_{i'_k}$ for $1 \leq k \leq n$ and $D_o \supseteq D_{o'}$. This property is called the *domain inclusion* of the parameters of shared actions. The intuition behind this condition comes from the fact that the output actions specify the method calls and the input ones specify the methods that can be called.

If the above conditions are satisfied between two interface automata A_1 and A_2, then we have to perform the renaming of parameter names in their pre and post-conditions in order to realize their composition.

Definition 2 (Parameter renaming) Given an action a in Shared(A_1, A_2), the signature of a is defined by a$(i_1,...,i_n) \rightarrow (o)$ in A_1 and by a$(i'_1,...,i'_n) \rightarrow (o')$ in A_2. The *renaming* of parameters in the semantics $\Psi_{A_1}(a)$ and $\Psi_{A_2}(a)$ is the substitution of i'_1 by i_1,..., i'_n by i_n, and o' by o in $\mathrm{Pre}_{\Psi_{A_2}(a)}$ and $\mathrm{Post}_{\Psi_{A_2}(a)}$ or the opposite in $\mathrm{Pre}_{\Psi_{A_1}(a)}$ and $\mathrm{Post}_{\Psi_{A_1}(a)}$.

We denote by $\Psi_{A_1/A_2}(a)$ and $\Psi_{A_2/A_1}(a)$, the semantics of a after the parameter renaming respectively in A_1 and A_2. We can now define properly the notion of the semantic compatibility of shared external actions.

Definition 3 (Semantic compatibility) Given an action a $\in$ Shared(A_1,A_2), if one of the following conditions is true, then the action a in A_1 is *semantically compatible* with the same action a in A_2 *i.e.* $\mathrm{SemComp}_a(A_1, A_2)$ is true (*otherwise* $\neg\mathrm{SemComp}_a(A_1, A_2)$ is true):

- if a $\in \Sigma^{\mathrm{O}}_{A_1} \wedge \mathrm{Pre}_{\Psi_{A_1/A_2}(a)} \Rightarrow \mathrm{Pre}_{\Psi_{A_2/A_1}(a)} \wedge \mathrm{Post}_{\Psi_{A_2/A_1}(a)} \Rightarrow \mathrm{Post}_{\Psi_{A_1/A_2}(a)}$,
- if a $\in \Sigma^{\mathrm{I}}_{A_1} \wedge \mathrm{Pre}_{\Psi_{A_2/A_1}(a)} \Rightarrow \mathrm{Pre}_{\Psi_{A_1/A_2}(a)} \wedge \mathrm{Post}_{\Psi_{A_1/A_2}(a)} \Rightarrow \mathrm{Post}_{\Psi_{A_2/A_1}(a)}$.

Definition 4 (Synchronized product $\otimes$) Given two composable interface automata A_1 and A_2, the *synchronized product* $A_1 \otimes A_2$ of A_1 and A_2 is defined by:

- $S_{A_1 \otimes A_2} = S_{A_1} \times S_{A_2}$ and $i_{A_1 \otimes A_2} = (i_{A_1}, i_{A_2})$; $\Sigma^{\mathrm{I}}_{A_1 \otimes A_2} = (\Sigma^{\mathrm{I}}_{A_1} \cup \Sigma^{\mathrm{I}}_{A_2}) \setminus \mathrm{Shared}(A_1, A_2)$*;*
- $\Sigma^{\mathrm{O}}_{A_1 \otimes A_2} = (\Sigma^{\mathrm{O}}_{A_1} \cup \Sigma^{\mathrm{O}}_{A_2}) \setminus \mathrm{Shared}(A_1, A_2)$*;* $\Sigma^{\mathrm{H}}_{A_1 \otimes A_2} = \Sigma^{\mathrm{H}}_{A_1} \cup \Sigma^{\mathrm{H}}_{A_2} \cup \{a \in \mathit{Shared}(A_1, A_2) \mid \mathit{SemComp}_a(A_1, A_2)\}$*;*
- $((s_1, s_2), a, (s'_1, s'_2)) \in \delta_{A_1 \otimes A_2}$ iff
  - $a \notin \mathrm{Shared}(A_1, A_2) \wedge (s_1, a, s'_1) \in \delta_{A_1} \wedge s_2 = s'_2$ *or* $a \notin \mathrm{Shared}(A_1, A_2) \wedge (s_2, a, s'_2) \in \delta_{A_2} \wedge s_1 = s'_1$ *or* $a \in \mathrm{Shared}(A_1, A_2) \wedge (s_1, a, s'_1) \in \delta_{A_1} \wedge (s_2, a, s'_2) \in \delta_{A_2} \wedge \mathrm{SemComp}_a(A_1, A_2)$*;*
- $\Psi_{A_1 \otimes A_1}$ is defined by:
  - Ψ_{A_1} for $a \in \Sigma_{A_2} \setminus \mathit{Shared}(A_1, A_2)$*;*
  - Ψ_{A_2} for $a \in \Sigma_{A_2} \setminus \mathit{Shared}(A_1, A_2)$*;*
  - $\langle \mathit{Pre}_{\Psi_{A_1}(a)}, \mathit{Post}_{\Psi_{A_2}(a)} \rangle$ for $a \in \mathit{Shared}(A_1, A_2) \cap \Sigma^{\mathrm{O}}_{A_1}$ such that $\mathrm{SemComp}_a(A_1, A_2)$*;*
  - $\langle \mathit{Pre}_{\Psi_{A_2}(a)}, \mathit{Post}_{\Psi_{A_1}(a)} \rangle$ for $a \in \mathit{Shared}(A_1, A_2) \cap \Sigma^{\mathrm{I}}_{A_1}$ such that $\mathrm{SemComp}_a(A_1, A_2)$*;*

The incompatibility between two interface automata A_1 and A_2 could happen due to (i) the existence of states (s_1,s_2) in the product $A_1 \otimes A_2$ such that there exists at least one action a in $\mathit{Shared}(A_1,A_2)$ enabled from s_1 and it is not from s_2 or inversely, or (ii) the action a is enabled from s_1 and s_2 but $\neg \mathit{SemComp}_a(A_1, A_2)$ is valid. These states are therefore illegal in the product $A_1 \otimes A_2$.

Definition 5 (Illegal states) The set of *illegal states*, denoted by $\mathrm{Illegal}(A_1,A_2) \subseteq S_{A_1} \times S_{A_2}$, in $A_1 \otimes A_2$ is defined by $\{(s_1, s_2) \in S_{A_1} \times S_{A_2} \mid (\exists\, a \in \mathrm{Shared}(A_1, A_2) \mid$ the condition $C_1 \oplus C_2$ is true[3] $)\}$.

$$C_1 = \begin{pmatrix} (a \in \Sigma^{\mathrm{O}}_{A_1}(s_1) \wedge a \notin \Sigma^{\mathrm{I}}_{A_2}(s_2)) \vee (a \in \Sigma^{\mathrm{O}}_{A_1}(s_1) \wedge a \in \Sigma^{\mathrm{I}}_{A_2}(s_2) \\ \wedge \neg \mathit{SemComp}_a(A_1, A_2)) \end{pmatrix}$$

$$C_2 = \begin{pmatrix} (a \in \Sigma^{\mathrm{O}}_{A_2}(s_2) \wedge a \notin \Sigma^{\mathrm{I}}_{A_1}(s_1)) \vee (a \in \Sigma^{\mathrm{O}}_{A_2}(s_2) \wedge a \in \Sigma^{\mathrm{I}}_{A_1}(s_1) \\ \wedge \neg \mathit{SemComp}_a(A_1, A_2)) \end{pmatrix}$$

Reaching states in $\mathit{Illegal}(A_1, A_2)$ is not sufficient to decide that A_1 and A_2 are incompatible (according to optimistic approach). Indeed, in this approach, if there is at least one environment that requests the appropriate input actions in $A_1 \otimes A_2$, and allows the no reachability of illegal states, then A_1 and A_2 can be assembled without producing deadlocks. The composition of A_1 and A_2, denoted by $A_1 || A_2$, is the restriction of their product to the set of states called *compatible*, denoted by $\mathit{Comp}(A_1,A_2)$. They are the states through which the interaction between the two components of A_1 and A_2 passes without having the risk of reaching illegal states by enabling only the locally controllable actions (input and hidden actions). The verification steps in this approach are similar to those described in [1], except that we consider the semantics of actions during the compatibility check by verifying the condition of semantic compatibility between the shared actions.

[3] $\oplus$ is XOR

4 Component Behavioral Contracts and Mismatches

The definitions of component interface mismatches [5, 2, 4] are essentially due to the reuse of components in a system design which is often harmed by mismatch cases such as: (i) names of exchanged messages between components do not correspond which may lead to deadlock situations, components regularly interact on the same action names; (ii) the orderings of messages or actions in both component protocols do not correspond; (iii) an action in a component that has no counterpart in the other one, or correspond to more than one action.

For component behavioral contracts specified with interface automata, the behavioral mismatch cannot be detected by applying the synchronized product between two composable interface automata as it was defined in Definition 4, because the case where there is no correspondence between the action names leads to them being absent from the set of shared actions. Thus, all of mismatched actions are interleaved asynchronously in the product. To avoid this constraint, our adaptation specification starts by establishing an abstract way to denote the composition requirements. We corroborate the explicit description of interactions between components thanks to *rules.* They relate the mismatched actions used in different components which are supposed to implement some interactions. Rules relate actions even if they do not really label some transitions in the automaton as required by the optimistic approach of interface automata.

The minimal adaptor specification of two interface automata A_1 and A_2 is the set of rules called a *mapping.* The mapping does not represent any behavioural detail about the adaptor.

Definition 6 (Rules and Mappings) *A* rule α *for two composable interface automata* A_1 *and* A_2*, is a pair* $\langle L_1, L_2 \rangle \in (2^{\Sigma^{\mathrm{O}}_{A_1}} \times 2^{\Sigma^{\mathrm{I}}_{A_2}}) \cup (2^{\Sigma^{\mathrm{I}}_{A_1}} \times 2^{\Sigma^{\mathrm{O}}_{A_2}})$[4] *such that* $(L_1 \cup L_2) \cap \mathit{Shared}(A_1, A_2) = \emptyset$ *and if* $|L_1| > 1$ *(or* $|L_2| > 1$*) then* $|L_2| = 1$ *(or* $|L_1| = 1$*);*

A mapping $\Phi(A_1, A_2)$ *for two composable interface automata* A_1 *and* A_2 *is a set of rules* α_i*, for* $1 \leq i \leq |\Phi(A_1, A_2)|$.

According to the above definition, a rule in our approach deals with one-to-one, many-to-one, and one-to-many correspondences between actions. More clearly, the adaptation may in general relate either an action or a group of actions of one automaton with one action in the other. For instance, a client authenticates itself by sending first its user name and then a password while the server accepts both data in a single login shot. We denote the set of the mismatched actions by $\mathit{Mismatch}_{\Phi}(A_1, A_2) = \{a \in \Sigma^{ext}_{A_1} \cup \Sigma^{ext}_{A_2} \mid \exists\, \alpha \in \Phi(A_1, A_2) \,.\, a \in \Pi_1(\alpha) \vee \mathrm{a} \in \Pi_2(\alpha)\}$[5].

Example 2. To illustrate the mapping relation, we define this latter between the actions of the interface automata Client and server as described in Figure 1 by: $\Phi(\mathit{Client},\mathit{Server}) = \{\langle \{\mathit{login}\}, \{\mathit{usr}, \mathit{pass}\}\rangle, \langle\{\mathit{finish}\}, \{\mathit{close}\}\rangle, \langle\{\mathit{ok}\}, \{\mathit{connected}\}\rangle$ $\langle\{\mathit{req},\mathit{arg}\}, \{\mathit{open}\}\rangle\ \}$. The set $\mathit{Shared}(A_1,\ A_2) = \{\mathit{error},\mathit{terminate}\}$.

[4] For some set S, 2^S is its power set.

[5] $\Pi_1(\langle a, b\rangle) = a$ and $\Pi_2(\langle a, b\rangle) = b$ are respectively the projection on the first element and the second element of the couple $\langle a, b\rangle$.

Given two composable interface automata A_1 and A_2 and a mapping $\Phi(A_1, A_2)$, if $\Phi(A_1, A_2) = \emptyset$, the adaptation of A_1 and A_2 has no sense and their synchronization is defined by their product $A_1 \otimes A_2$ as it was defined in section 3. Otherwise, we proceed on two steps: (i) we check first the semantic adaptability between the mismatched actions in the mapping $\Phi(A_1, A_2)$. (ii)if the semantic adaptability check was successfully made without giving rise to incompatibilities, we generate the adaptor of A_1 and A_2 according to the mapping $\Phi(A_1, A_2)$. If the generated adaptor is non-empty and it is compatible with both of A_1 and A_2, we say that A_1 and A_2 are *adaptable*.

5 Component Adaptation

In this section we present our approach to adapt components specified with behavioral contracts.

5.1 Semantic Adaptability

The semantic adaptability between the mismatched actions of two composable interface automata has to be made before generating the adaptor. The mismatched actions have to respect some constraints at the level of their semantics. Let us consider two interface automata A_1 and A_2 and a given mapping $\Phi(A_1, A_2)$. To perform the semantic adaptability check between A_1 and A_2 according to $\Phi(A_1, A_2)$, it is required that for each rule $\alpha = \langle L_1, L_2 \rangle \in \Phi(A_1, A_2)$ the following conditions hold:

1. $\sum_{a \in L_1} \mid P_a^{\mathrm{i}} \mid = \sum_{b \in L_2} \mid P_b^{\mathrm{i}} \mid$;
2. $\sum_{a \in L_1} \mid P_a^{\mathrm{o}} \mid = \sum_{b \in L_2} \mid P_b^{\mathrm{o}} \mid$;
3. if $|L_1| = 1$ and $\mid L_2 \mid \geq 1$ where $L_1 = \{a\}$, $L_2 = \{b_1, ..., b_{|L_2|}\}$, and $P_a^{\mathrm{o}} = \{o_a\}$ then there exists exactly one action $b_k \in L_2$ $(1 \leq k \leq \mid L_2 \mid)$ such that $P_{b_k}^{\mathrm{o}} = \{o_{b_k}\}$, $P_{b_l}^{\mathrm{o}} = \emptyset$ for $1 \leq l \leq \mid L_2 \mid$ and $l \neq k$, and the two output parameters o_a and o_{b_k} have to satisfy the domain inclusion condition:
   - if $L_1 \subseteq \Sigma_{A_1}^{\mathrm{O}}$, then $D_{o_a} \subseteq D_{o_{b_k}}$;
   - else $D_{o_a} \supseteq D_{o_{b_k}}$;

   θ_α denotes the tuple (a,b_k). If $P_a^{\mathrm{o}} = \{\}$, (a,b_k) is not defined;
4. the condition is analogous to the previous one with $|L_1| \geq 1$ and $|L_2| = 1$ where $L_1 = \{a_1, ..., a_{|L_1|}\}$ and $L_2 = \{b\}$;
5. there exists a function $\varphi_\alpha^{\mathrm{i}} : \bigcup_{a \in L_1} P_a^{\mathrm{i}} \to \bigcup_{b \in L_2} P_b^{\mathrm{i}}$ that associates each input parameter p of actions in L_1 with an input parameter q of actions in L_2. The function $\varphi_\alpha^{\mathrm{i}}$ have to satisfy the domain inclusion condition:
   - if $L_1 \subseteq \Sigma_{A_1}^{\mathrm{O}}$, then $D_p \subseteq D_{\varphi_\alpha^{\mathrm{i}}(p)}$ where $p \in \bigcup_{a \in L_1} P_a^{\mathrm{i}}$;
   - else $D_{\varphi_\alpha^{\mathrm{i}}(p)} \subseteq D_p$ where $p \in \bigcup_{a \in L_1} P_a^{\mathrm{i}}$.

The first and the second conditions state that the number of input (respectively output) parameters of actions in L_1 is equal to the number of input (respectively output) parameters of actions in L_2. The third condition states the relations between the output parameter of the action $a \in L_1$ and the one of the action $b_k \in L_2$. We assume that the other actions in $L_2 \setminus \{b_k\}$ have no output parameters. The intuition behind these conditions is to avoid conflicts between the pre and post-conditions during the semantic adaptability check by ensuring the equality between the number of input and output parameters.

The renaming of the input and output parameter in the semantics of actions in $\mathit{Mismatch}_{\Phi}(A_1, A_2)$ is defined as follows. For all $a \in L_1$ and $b \in L_2$, the parameter renaming is defined by the substitution of each input parameter i of a in $\mathit{Pre}_{\Psi_{A_1}(a)}$ and $\mathit{Post}_{\Psi_{A_1}(a)}$ by $\varphi^{\mathrm{i}}_{\alpha}(i)$ or the substitution of each input parameter i' of b in $\mathit{Pre}_{\Psi_{A_2}(b)}$ and $\mathit{Post}_{\Psi_{A_2}(b)}$ by ${\varphi^{\mathrm{i}}_{\alpha}}^{-1}(i')$[6]. If the couple $\theta_\alpha = (a, b)$ exists, the parameter renaming is defined by the substitution of the output parameter o_a in $\mathit{Post}_{\Psi_{A_1}(a)}$ by o_b or the substitution of the output parameter o_b in $\mathit{Post}_{\Psi_{A_2}(b)}$ by o_a.

We denote by $\Psi_{A_1/\alpha}(a)$ and $\Psi_{A_2/\alpha}(b)$ respectively the semantics of actions in $\Pi_1(\alpha)$ and actions in $\Pi_2(\alpha)$ after the parameter renaming.

Definition 7 (Semantic Adaptability) Given two composable interface automata A_1 and A_2 and an adaptation mapping $\Phi(A_1, A_2)$ such that the conditions 1, 2, 3, 4, and 5 introduced in Section 5.1 are satisfied, the *semantic adaptability* $\mathrm{SemAdap}_\alpha(A_1, A_2)$ of a rule α in $\Phi(A_1, A_2)$ is satisfied iff the following conditions are fulfilled:

1. If $\Pi_1(\alpha) \subseteq \Sigma^{\mathrm{O}}_{A_1}$, then

$$\left(\begin{array}{c} \bigwedge_{a \in \Pi_1(\alpha)} \mathit{Pre}_{\Psi_{A_1/\alpha}(a)} \Rightarrow \bigwedge_{b \in \Pi_2(\alpha)} \mathit{Pre}_{\Psi_{A_2/\alpha}(b)} \\ \wedge \\ \bigwedge_{a \in \Pi_1(\alpha)} \mathit{Post}_{\Psi_{A_1/\alpha}(a)} \Leftarrow \bigwedge_{b \in \Pi_2(\alpha)} \mathit{Post}_{\Psi_{A_2/\alpha}(b)} \end{array} \right)$$

2. If $\Pi_1(\alpha) \subseteq \Sigma^{\mathrm{I}}_{A_1}$, then the condition is analogous to the previous one by inversing the implications.

We say that A_1 and A_2 are semantically adaptable *according to the mapping $\Phi(A_1,\ A_2)$ if the semantic adaptability of each rule $\alpha \in \Phi(A_1,\ A_2)$ holds.*

The semantic adaptability conditions are stated in a similar way as the semantic compatibility of the shared actions defined in Definition 3 except that for adaptation, we treat sets of mismatched actions associated by the rules of the mapping.

Example 3. To illustrate mismatches between actions belonging to two behavioral contracts, we consider the two composable interface automata *Client* and *Server*, that specify component behavioral contracts, shown in Figure 1 and a mapping $\Phi(\mathit{Client},\mathit{Server})$ as defined in Example 2.

The mismatched actions are described and classified by the rules in Table 1. The function $\varphi^{\mathrm{i}}_{\alpha_2}$ is defined by $\{\mathit{msg} \mapsto \mathit{logmsg}\}$. The function $\varphi^{\mathrm{i}}_{\alpha_4}$ is not defined. The function $\varphi^{i}_{\alpha_1}$ is defined by and $\{\mathit{uname} \mapsto \mathit{username},\ \mathit{lu} \mapsto \mathit{lengthu},$

[6] f^{-1} is the inverse function of *f*.

Table 1. The signatures of actions in *Mismatch*$_\Phi$(*Client*,*Server*)

	Client	***Server***
α_1	login(uname,passwd,lu,lp)→(exist)	usr(username,lengthu)→()
		pass(password,lengthp)→(exist)
α_2	ok(msg)→()	connected(logmsg)→()
α_3	req(read)→()	open(readonly,filename)→(open)
	arg(file)→(status)	
α_4	finish()→(status)	close()→(closed)

Table 2. The semantics of actions in *Mismatch*$_\Phi$(*Client*,*Server*)

Client	***Server***
$Pre_{\Psi_{Client}(login)} \equiv 1 < \text{lu} \leq 20 \wedge 8 \leq \text{lp} \leq 10$ $Post_{\Psi_{Client}(login)} \equiv \text{exist} = 1 \vee \text{exist} = 0$	$Pre_{\Psi_{Server}(usr)} \equiv 1 < \text{lengthu} \leq 30$ $Post_{\Psi_{Server}(usr)} \equiv \text{true}$
	$Pre_{\Psi_{Server}(pass)} \equiv 6 \leq \text{lengthp} \leq 10$ $Post_{\Psi_{Server}(pass)} \equiv \text{exist} = 1 \vee \text{exist} = 0$
$Pre_{\Psi_{Client}(ok)} \equiv \text{true}$ $Post_{\Psi_{Client}(ok)} \equiv \text{true}$	$Pre_{\Psi_{Server}(connected)} \equiv \text{true}$ $Post_{\Psi_{Server}(connected)} \equiv \text{true}$
$Pre_{\Psi_{Client}(req)} \equiv \text{read} = 0 \vee \text{read} = 1$ $Post_{\Psi_{Client}(req)} \equiv \text{true}$	$Pre_{\Psi_{Server}(open)} \equiv \text{readonly} = 0 \vee \text{readonly} = 1$ $Post_{\Psi_{Server}(open)} \equiv \text{open} = 0 \vee \text{open} = 1$
$Pre_{\Psi_{Client}(arg)} \equiv \text{true}$ $Post_{\Psi_{Client}(arg)} \equiv \text{status} = 0 \vee \text{status} = 1$	
$Pre_{\Psi_{Client}(finish)} \equiv \text{true}$ $Post_{\Psi_{Client}(finish)} \equiv \text{status} = 0 \vee \text{status} = 1$	$Pre_{\Psi_{Server}(close)} \equiv \text{true}$ $Post_{\Psi_{Server}(close)} \equiv \text{closed} = 0 \vee \text{closed} = 1$

passwd↦*password*, *lp*↦*lengthp*}. The function $\varphi^{\mathrm{i}}_{\alpha_3}$ is defined by {*read*↦*readonly*, *file*↦*filename*}. The function $\varphi^{\mathrm{i}}_{\alpha_4}$ is empty. $\theta_{\alpha_1} = (login,pass)$, θ_{α_2} is not defined, $\theta_{\alpha_3} = (arg,open)$, and $\theta_{\alpha_4} = (finish,close)$. The parameters *uname*, *passwd*, *username*, *password*, *msg*, *logmsg*, *file*, and *filename* are strings. The parameters *lu*, *lp*, *lengthu*, *lengthp*, *read*, *readonly*, *status*, *open*, and *closed* are integers. As the reader can conclude, the conditions to perform the semantic adaptability check hold for all α in $\Phi(A_1, A_2)$:

- for all $\alpha \in \Phi(A_1, A_2)$, $\sum_{a\in\Pi_1(\alpha)} |P^{\mathrm{i}}_a| = \sum_{b\in\Pi_2(\alpha)} |P^{\mathrm{i}}_b|$ and $\sum_{a\in\Pi_1(\alpha)} |P^{\mathrm{o}}_a| = \sum_{b\in\Pi_2(\alpha)} |P^{o}_b|$;
- the domain inclusion conditions are satisfied for θ_* and φ^i_* where $* \in \Phi($*Client*, *Server*$)$.

The semantics of the mismatched actions respectively for *Client* and *Server* are listed in Table 2. After unifying the mismatched actions in *Mismatch*$_\Phi$(*Client*, *Server*), the reader can easily verify the semantic adaptability for all α in Φ(*Client*, *Server*) holds. For example, for the rule α_1,

$Pre_{\Psi_{Client/\alpha_1}(login)} \Rightarrow (Pre_{\Psi_{Server/\alpha_1}(usr)} \wedge Pre_{\Psi_{Server/\alpha_1}(pass)})$ is satisfiable $((1 < \text{lu} \leq 20 \wedge 8 \leq \text{lp} \leq 10) \Rightarrow (1 < \text{lu} \leq 30 \wedge 6 \leq \text{lp} \leq 10))$. Also, $Post_{\Psi_{Client/\alpha_1}(login)} \Leftarrow (Post_{\Psi_{Server/\alpha_1}(usr)} \wedge Post_{\Psi_{Server/\alpha_1}(pass)})$ is satisfiable $((\text{exist} = 1 \vee \text{exist} = 0) \Leftarrow (\text{true} \wedge (\text{exist} = 1 \vee \text{exist} = 0)))$. We can deduce that *Client* and *Server* are semantically adaptable according to $\Phi(Client, Server)$.

5.2 Adaptor Specification and Construction

After verifying the semantic adaptability between two composable interface automata A_1 and A_2 according to a mapping $\Phi(A_1, A_2)$, we treat in this section the interface automaton specification and construction of their adaptor. The adaptor must be composable with A_1 and A_2, and must also satisfy the mapping rules and respect the component protocols specified by A_1 and A_2.

Definition 8 (Adaptor) *Given two composable interface automata A_1, A_2, and a mapping $\Phi(A_1, A_2)$, an* adaptor *for A_1 and A_2 according to the mapping $\Phi(A_1, A_2)$ is an interface automaton $Ad = \langle\ S_{Ad},\ I_{Ad},\ \Sigma^{\mathrm{I}}_{Ad},\ \Sigma^{\mathrm{O}}_{Ad},\ \Sigma^{\mathrm{H}}_{Ad},\ \delta_{Ad}\ \rangle$ such that*

- $\Sigma^{\mathrm{I}}_{Ad} = \{a \mid a \in Mismatch_{\Phi}(A_1, A_2) \cap (\Sigma^{\mathrm{O}}_{A_1} \cup \Sigma^{\mathrm{O}}_{A_2})\}$;
  - . For all a $\in \Sigma^{\mathrm{I}}_{Ad}$, $\Psi_{Ad}(a) = \Psi_{A_1}(a)$ if a $\in \Sigma^{\mathrm{O}}_{A_1}$. Otherwise, $\Psi_{Ad}(a) = \Psi_{A_2}(a)$;
- $\Sigma^{\mathrm{O}}_{Ad} = \{a \mid a \in Mismatch_{\Phi}(A_1, A_2) \cap (\Sigma^{\mathrm{I}}_{A_1} \cup \Sigma^{\mathrm{I}}_{A_2})\}$;
  - . For all a $\in \Sigma^{\mathrm{O}}_{Ad}$, $\Psi_{Ad}(a) = \Psi_{A_1}(a)$ if a $\in \Sigma^{\mathrm{I}}_{A_1}$. Otherwise, $\Psi_{Ad}(a) = \Psi_{A_2}(a)$;
- $\Sigma^{\mathrm{H}}_{Ad} \subseteq \{\epsilon\}$; *in the adaptor this set represents the internal actions that do nothing, which are associated to input/output actions in mismatched components which are not concerned with the mapping (the adaptation);*
- $\delta_{Ad} \subseteq S_{Ad} \times \Sigma^{\mathrm{I}}_{Ad} \cup \Sigma^{\mathrm{O}}_{Ad} \cup \{\epsilon\} \times S_{Ad}$;
- $Shared(Ad, A_1) = \bigcup_{\alpha \in \Phi(A_1, A_2)} \Pi_1(\alpha)$; $Shared(Ad, A_2) = \bigcup_{\alpha \in \Phi(A_1, A_2)} \Pi_2(\alpha)$;

The adaptor must satisfy the following condition in order to ensure that the mapping rules are respected, therefore the mismatch between components is resolved.

The condition on the adaptor paths: For all execution path $\sigma = s_1 a_1 s_2 a_2 ... s_i a_i ... s_n$ in the adaptor Ad , such that $s_i \in S_{Ad}$ and $a_i \in \Sigma^{\mathrm{O}}_{Ad} \cup \Sigma^{\mathrm{I}}_{Ad}$, if $\exists \alpha \in \Phi(A_1, A_2)$ such that the output actions (enabled as input in Ad) of α are present in σ then they are succeed, in σ, by there correspondent input actions (enabled as output in Ad).

Property 1 *An adaptor* Ad *for two interface automata A_1 and A_2 according to a mapping $\Phi(A_1, A_2)$ is composable with A_1 and A_2.*

The property can be easily verified according to Definition 8. Indeed, by considering the set of actions of Ad, Σ^{I}_{Ad} and Σ^{O}_{Ad}, the condition of composability, as defined in Section 4, can be easily verified with the set of actions of A_1 and A_2.

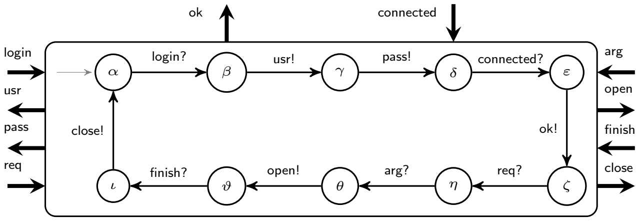


Fig. 2. The adaptor *Adaptor* for *Client* and *Server*

The composition of A_1 and A_2 is performed by synchronizing first *Ad* with either A_1 or A_2, computing their composition according to our extended approach, and then by composing the resulting composition with the remaining automaton. We suppose that the actions of the adaptor have the same signatures and semantics as actions in $Mismatch_{\Phi}(A_1,A_2)$. If the composite interface automaton $A_1 \parallel Ad \parallel A_2$ is non empty then A_1 and A_2 are compatible after their adaptation at the protocol and the semantic levels.

To generate the adaptor *Ad* from A_1, A_2 , and the mapping $\Phi(A_1, A_2)$, we have to explore in parallel the states and the transitions of both automata A_1 and A_2. For the lack of space, the details of the algorithm to perform adaptor generation is not described in this paper, however this algorithm is the same as our algorithm in [6] that constructs an adaptor for two composable interface automata A_1, A_2, and a given non empty mapping $\Phi(A_1, A_2)$. In fact the contribution of this paper compared to the approach in [6] is the handling of action semantics in component adaptability thanks to the design of components with behavioral contracts. The step for generating the interface automaton of the adaptor comes after verifying the semantic compatibility between A_1 and A_2. However in [6] we considered only the protocol level in the adaptation. So, the algorithm is basically a loop which reads in parallel A_1 and A_2 and constructs as one goes along the set of states and the set of transitions of the adaptor. The algorithm is executed by respecting the reordering of events of both interfaces A_1 and A_2. The algorithm marks and removes from the generated graph all the fragments of paths that do not respect the condition on the adaptor paths.

The part of the algorithm that constructs the set of states and transitions has the time complexity $\mathbf{O}(|S_{A_1} \times S_{A_2}|.(|\delta_{A_1}| + |\delta_{A_2}|))$. The time complexity of the part that removes the undesired path fragments is linear in the number of the generated states.

Example 4. As the reader can conclude, *Adaptor* is composable with both *Client* and *Server* presented in Example 1 and it satisfies all the items of Definition 8. Our proposed algorithm in [6] generates exactly the same interface automaton shown in Figure 2. Suppose that the semantic compatibility between the shared actions *error* and *terminate* holds, then *Adaptor* is compatible with both *Client* and *Server*. The composite interface automaton (*Client* $\parallel$ *Adaptor*) $\parallel$ *Server* is non empty which makes *Client* and *Server* compatible after their adaptation.

6 Related Work

Several techniques of adaptation show how to automatically derive adaptors in order to eliminate mismatches between components during their interactions. In [13], the authors propose an interesting approach based on finite state machines to adapt components specified by interfaces describing component protocol and action signatures. This approach deals with one-to-one relations between actions. In [8], the authors propose the Smart Connectors approach which allows the construction of adaptors using the provided and required interfaces of the components in order to resolve partial matching problems in COTS component acquisition. In [2], the authors have proposed a formal approach based on calculus to generate automatically adaptors using the Prolog language. The authors in [3] present an approach based on session types, exploited to specify component behaviors, to adapt heterogeneous components that may present mismatching interaction behaviors. In [7], Hemer has proposed, using template from the CARE language, to define adaptation strategies for modifying and combing components. In [9], the authors have proposed a model of adaptors expressed in the B formal method, allowing to define the interoperability between components. In [11] the authors introduce the concept of parameterized contracts and a model for component interfaces, they also present algorithms and tools for specifying and analyzing component interfaces in order to check interoperability and to generate adapted component interfaces. In [12], the authors propose to generate semi-automatically adaptors, at the protocol level, for concurrent components that are specified with finite state machines. Another approach that deals with the adaptation of component at the protocol level is presented in [10]. The authors proposed an algorithmic approach for checking whether incompatible interaction protocols of component interfaces can be made compatible by inserting a protocol converter between them. The approaches described above propose solutions for the component adaptation based on different specification formalisms of component interfaces. Our approach is different from the others, because we propose a solution to adapt particular components that are specified by interface automata. This formalism allows to exploit optimistic approach [1] to check to component interoperability. This adaptation approach deals with the signature, the semantic, and the protocol levels, and deals also with possibly complex adaptation scenarios : one-to-one and one-to-many correspondences between actions.

7 Conclusion

In this paper, we proposed a formal approach for the automatic development of component adaptors, allowing the elimination of mismatches between interacting components. Our component interfaces are described with behavioral contracts, which allow to handle all component facets for their adaptation, by considering component informations at levels of action signatures, their semantics, and their protocols. So, we proposed a formal framework for component adaptation, based on the following concepts: behavioral contracts, their composability, their synchronization, and their semantic compatibility. Therefore we specified these contracts with interface automata enriched by the action semantics. We exploited the obtained formalism to adapt the interface automata approach to verify compatibility between components specified with behavioral contracts. When components are incompatible due to mismatches, we proposed to specify a correspondence mapping between the mismatched actions of two components as a first abstract specification of the adaptor. This mapping deals with one-to-one and one-to-many correspondences between the actions. Finally, we proposed an approach that generates the adaptor for two composable interface automata according to a fixed mapping. The generated adaptor allows to eliminate mismatches at signature, semantic, and protocol levels.

References

1. Alfaro, L., Henzinger, T.A.: Interface automata. ACM Press, 9th Annual Symposium of FSE (Foundations of Software Engineering) pp. 109–120 (2001)
2. Bracciali, A., Brogi, A., Canal, C.: A formal approach to component adaptation. Journal of Systems and Software **74**, 45–54 (2005)
3. Brogi, A., Canal, C., Pimentel, E.: Behavioural types and component adaptation. In: Algebraic Methodology and Software Technology: 10th International Conference, AMAST 2004, volume 3116 / 2004 of LNCS. pp. 42–56. Springer-Verlag GmbH (2004)
4. Canal, C., Murillo, J., Poizat, P.: Software adaptation. Special Issue on Software adaptation **12**(1), 9–31 (2006)
5. Canal, C., Poizat, P., Sala*ü*n, G.: Synchronizing behavioural mismatch in software composition. Proc. of FMOODS'06, LNCS **6**, 63–77 (2006)
6. Chouali, S., Mouelhi, S., Mountassir, H.: Adapting component behaviours using interface automata. IEEE Computer Society proceedings, Euromicro SEAA 2010 conference (September 2010)
7. Hemer, D.: A formal approach to component adaptation and composition. In Proceedings of the Twenty-eighth Australasian conference on Computer Science ACSC '05 Newcastle, Australia pp. 259–266 (2005)
8. Min, H., Choi, S., Kim, S.: Using smart connectors to resolve partial matching problems in cots component acquisition. LNCS, Springer-Verlag, Berlin, Germany **3054**, 40–47 (2004)
9. Mouakher, I., Lanoix, A., Souquières, J.: Component Adaptation: Specification and Verification. In: 11th International Workshop on Component Oriented Programming (WCOP 2006). p. 8. ECOOP 2006, Nantes, France (07 2006)
10. Passerone, R., de Alfaro, L., Henzinger, T.A., Sangiovanni-Vincentelli, A.L.: Convertibility verification and converter synthesis: two faces of the same coin [ip block interfaces]. In: IEEE/ACM ICCAD 2002. pp. 132–139 (Nov 2002). https://doi.org/10.1109/ICCAD.2002.1167525
11. Reussner, R.: Automatic component protocol adaptation with the coconut/j tool suite. Future Generation Computer Systems **19**(5), 627–639 (2003)
12. Schmidt, H., Reussner, R.: Generating adaptors for concurrent component protocol synchronisation. In the proceeding of the Fifth IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems pp. 213–229 (2002)
13. Yellin, D., Strom, R.: Protocol specifications and components adaptors. ACM Transactions on Programming Languages and Systems **19**(2), 292–333 (1997)