Adapting Components Behaviours using Interface Automata

Samir Chouali, Sebti Mouelhi, Hassan Mountassir

Adapting Component Behaviours using Interface Automata

Samir Chouali, Sebti Mouelhi, Hassan Mountassir
Laboratory of Computer Science, LIFC
University of Franche Comté, Besançon, FRANCE
Email: {schouali,smouelhi,hmountassir}@lifc.univ-fcomte.fr

Abstract

One of the principal goal of Component-Based Software Engineering (CBSE) is to allow the reuse of components in diverse situations without affecting their codes. To reach this goal, it is necessary to propose approaches to adapt a component with its environment when behavioural mismatches occur during their interactions. In this paper, we present a formal approach based on interface automata to adapt components in order to eliminate possible behavioural mismatches, and then insure more flexible interoperability between components.

1. Introduction

The idea of component based software engineering [13, 5] is to develop software applications not from scratch but by assembling various library components. This development approach allows software reuse without changing components code. A component is a unit of composition with contractually specified interfaces and explicit dependencies [13]. An interface may describe component information at the level of signature (method names and their types), behavior or protocol (scheduling of method calls), semantic (method semantics) without disclosing the component implementation. The success of applying the component based approach depends on the interoperability which holds between two components when their interfaces are compatible. Usually interoperability is not guaranteed when we reuse and assemble components. This is due to possible mismatches that may occur, between components, at different levels cited above [4]. In this case, components adaptation should be performed in order to eliminate the resulting mismatches.

In this paper [1], we focus on adapting components, described by interfaces which are specified by *interface automata* [1]. This formalism has the ability to model a temporal order of, both the input requirements (input actions), the output behavior (output actions) of a component, and the local events of a component. The composition of two interfaces is achieved by synchronizing their shared output and input actions. An interesting verification approach was also proposed in [1], to detect interface incompatibilities that may occur when, from some states in the synchronized product, one automaton issues a shared action as output which is not accepted as input in the other. We say that those states are *illegal*. Our purpose is to generate automatically an adaptor (*interface in the middle*) for exactly two component interface automata according to a mapping that establishes a number of rules relating the component actions. It allows the elimination of mismatches at the signature and the protocol levels.

One of the aims of this paper is to bring together the notion of adaptability of interface automata and their optimistic approach of composition. The latter makes the two interface automata A_1 and A_2 compatible if there is an environment that prevents their synchronization to enter illegal states. The composition approach of the other automata-based interface formalisms are considered pessimistic.

2. Interface Automata

Interface automata have been introduced by L. Alfaro and T. Henzinger [1], to model the temporal behavior of software component interfaces. Every component interface is described by one interface automaton.

Definition 1 (Interface Automata). *An* interface automaton $A = \langle S_A, I_A, \Sigma_A^I, \Sigma_A^O, \Sigma_A^H, \delta_A \rangle$ *consists of: a finite set* S_A *of states*; *a subset of initial states* $I_A \subseteq S_A$. *Its cardinality card*$(I_A) \leq 1$ *and A is called* empty *if* $I_A = \emptyset$; *three disjoint sets* Σ_A^I, Σ_A^O *and* Σ_A^H *of inputs, output, and hidden actions*; *a set* $\delta_A \subseteq S_A \times \Sigma_A \times S_A$ *of transitions between states*.

In this article, we use the formalism of interface automata (IAs) defined by L. Alfaro and T. Henzinger without

[1]This work has received support from the French National Agency for Research , ANR-06-SETI-017 (TACOS).

any extension, except that the set of hidden actions of an interface may contain the special action *epsilon* ϵ that symbolizes the no-operation event. We redefine, as consequence, the composability condition of interface automata. Two interface automata A_1 and A_2 are *composable* if $\Sigma^I_{A_1} \cap \Sigma^I_{A_2} = \Sigma^O_{A_1} \cap \Sigma^O_{A_2} = \Sigma^H_{A_1} \setminus \{\epsilon\} \cap \Sigma_{A_2} = \Sigma^H_{A_2} \setminus \{\epsilon\} \cap \Sigma_{A_1} = \emptyset$. All the notations used and the specificities of the optimistic view of interface automata composition are detailed in the reference article of the formalism [1].

3. Behavioural Mismatch in IAs

The behavioural mismatch of IAs cannot be detected by applying the synchronized product between two composable interface automata as it was defined in [1], because the case where the action names do not correspond leads to make them absent in the set of shared actions. Thus, all of mismatched actions do not synchronize and they are interleaved asynchronously in the product. To solve this constraint, we define a set of rules called a *mapping* which provides the minimal specification of the adaptor of two adaptable components. These rules define correspondences between the mismatched actions.

Definition 2 (Rules and Mappings). *A* rule α *for two composable interface automata* A_1 *and* A_2*, is a tuple* $\langle L_1, L_2 \rangle \in (2^{\Sigma^O_{A_1}} \times 2^{\Sigma^I_{A_2}}) \cup (2^{\Sigma^I_{A_1}} \times 2^{\Sigma^O_{A_2}})$ *such that* $(L_1 \cup L_2) \cap Shared(A_1, A_2) = \emptyset$ *and if* $\mid L_1 \mid > 1$ (*or* $\mid L_2 \mid > 1$) *then* $\mid L_2 \mid = 1$ (*or* $\mid L_1 \mid = 1$). *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* $1 \leq i \leq \mid \Phi(A_1, A_2) \mid^2$.

According to the Definition 2, a rule in our approach deals with one-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. The adaptability of interface automata permits to make sense to the synchronization of some non-shared actions between two composable interface automata A_1 and A_2 according to a fixed mapping $\Phi(A_1, A_2)$. We denote the set of the mismatched actions by $Mismatch_\Phi(A_1, A_2) = \{a \in \Sigma^{ext}_{A_1} \cup \Sigma^{ext}_{A_2} \mid \exists\, \alpha \in \Phi(A_1, A_2)\ .\ a \in \Pi_1(\alpha) \vee a \in \Pi_2(\alpha)\}$[3].

We call that two interface automata A_1 and A_2 are *adaptable* if they are composable, their mapping $\Phi(A_1, A_2)$ is not empty, and there exists an adaptor for them which is *compatible* with both A_1 and A_2. We distinguish two principle cases: (i) if the generated adaptor is empty then A_1 and A_2 cannot be adapted (ii) Otherwise, if $A_1 \parallel Ad \parallel A_2$ is non empty, then A_1 and A_2 are compatible after their adaptation.

[2] $\mid S \mid$ is the cardinality of some set *S*.

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

4. Adaptor Specification

We present the adaptor specification of two composable interface automata A_1 and A_2. Some preliminaries have to be introduced before. Given an interface automata *A*, we denote by $\Theta^S_A(s) \subseteq S^*_A$ the set of successor finite runs $\theta = s_1 a_1 s_2 a_2 ... s_n$ such that $s_1 = s$, s_n is the initial state or a state that have no outgoing transitions, and for all $1 \leq i < n$, there is a transition $(s_i, a_i, s_{i+1}) \in \delta_A$. We denote by $\Theta^P_A(s) \subseteq S^*_A$ the set of predecessor finite runs $\theta = s_1 a_1 s_2 a_2 ... s_n$ is defined exactly as $\Theta^S_A(s)$ except $s_1 = i$ where $i \in I_A$ and $s_n = s$. The set Θ_A of all finite runs of A equals to $\Theta^S_A(i)$ where $i \in I_A$. We say that a succession of transitions $s_1 a_1 s_2 a_2 ... s_n$ (for $n \geqslant 2$) is included in a run σ in $\Theta^S_A(s)$ or $\Theta^P_A(s)$ (denoted by the operator $\sqsubseteq$), if all transitions of $s_1 a_1 s_2 a_2 ... s_n$ are transitions of σ.

Definition 3 (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 automata* $Ad = \langle S_{Ad}, I_{Ad}, \Sigma^I_{Ad}, \Sigma^O_{Ad}, \Sigma^H_{Ad}, \delta_{Ad} \rangle$ *such that*

- $\Sigma^I_{Ad} = \{a \in \Sigma^O_{A_1} \cup \Sigma^O_{A_2} \mid a \in Mismatch_\Phi(A_1, A_2)\}$; $\Sigma^O_{Ad} = \{a \in \Sigma^I_{A_1} \cup \Sigma^I_{A_2} \mid a \in Mismatch_\Phi(A_1, A_2)\}$; $\Sigma^H_{Ad} \subseteq \{\epsilon\}$; $\delta_{Ad} \subseteq S_{Ad} \times \Sigma^I_{Ad} \cup \Sigma^O_{Ad} \cup \{\epsilon\} \times S_{Ad}$;
- $Shared(Ad, A_1) = \bigcup_{\alpha \in \Phi(A_1, A_2)} \Pi_1(\alpha)$; $Shared(Ad, A_2) = \bigcup_{\alpha \in \Phi(A_1, A_2)} \Pi_2(\alpha)$;
- *For all* $s \in S_{Ad}$ *and* $\sigma \in \Theta^P_{Ad}(s)$*, if there exist* $r_1 a_1 ... r_n a_n s \sqsubseteq \sigma$ *and* $\alpha \in \Phi(A_1, A_2)$ *such that* $\Pi_i(\alpha) \subseteq \Sigma^O_{A_i}$ *for* $i \in \{1, 2\}$ *and* $\Pi_i(\alpha) \subseteq \bigcup_{k \in 1..n} \{a_k\}$*, then for all* $\rho \in \Theta^S_{Ad}(s)$*, there exists* $s b_1 ... b_m t_m \sqsubseteq \rho$ *such that* $\Pi_{3-i}(\alpha) \subseteq \Sigma^I_{A_{3-i}}$ *and* $\Pi_{3-i}(\alpha) \subseteq \bigcup_{l \in 1..m} \{b_l\}$.

Property 1 An adaptor *Ad* for two composable 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 3 and the definition of the interface automata composability.

Example 1. Given the two composable interface automata *Client* and *Server* shown in Figure 1 and a mapping Φ(*Client*,*Server*) = {$\langle$ {*login*!}, {*usr*?,*pass*?}$\rangle$, $\langle$ {*req*!}, {*query*?}$\rangle$, $\langle$ {*arg*!}, {*value*?}$\rangle$, $\langle$ {*ack*?}, {*service*!}$\rangle$ $\langle$ {*ok*?}, {*connected*!}$\rangle$}. As the reader can conclude, *Client* and *Server* are adaptable since their adaptor *Ad* (cf. Figure 1) is composable and compatible with both of them and it satisfies all the items of Definition 3. The composite interface automaton (*Client* $\parallel$ *Adaptor*) $\parallel$ *Server* is non empty.

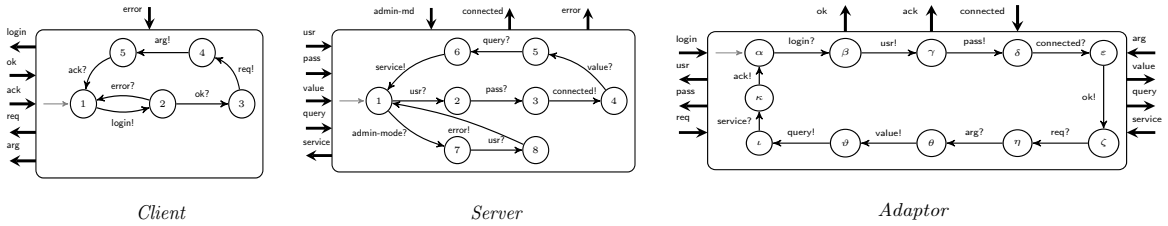

Figure 1. The adaptor Ad for a variant of a client/server system

Consequently, *Client* and *Server* are compatible after their adaptation by *Adaptor*. Our proposed algorithm presented in section 5 is supposed to generate the same interface automaton as *Adaptor*.

5. Adaptor Construction

The adaptor construction is splited into two algorithms. The result of the first one (Algorithm 1) is illustrated in Figure 2. It traverses the two interface automata A_1 and A_2 to be adapted, according to a mapping $\Phi(A_1, A_2)$, by exploring alternatively the states and the transitions as far as possible along the branches of each of them. It updates as one goes along the set of states S and the set of transitions T initially set to the empty set. Red states represents states that from where the output actions of a given rule α in the mapping are not followed by all their corresponding input ones. These states are called *blocking*.

To satisfy the last condition of Definition 3, Algorithm 2 removes all the fragments of runs leading to the blocking states. It traverses the resulting graph of Algorithm 1 using a loopback starting from blocking states until states which have at least one outgoing transition leading to a non marked state by the loop. If there is no remaining state then the adaptor is not defined between A_1 and A_2. We introduce the set $PRED^d_T(B)$ ($B \subseteq S$) by $\{\ \lambda \in S \backslash B.\ \lambda \in Pred_T(\mu) \wedge \mu \in B \wedge (deg^o_T(\lambda)^4 = 1 \vee (deg^o_T(\lambda) > 1 \wedge Succ_T(\lambda) \subseteq B))\}$ where $Pred_T(\lambda)$ and $Succ_T(\lambda)$ are respectively the sets of predecessor and successor of $\lambda \in S$ in T .

Algorithm 2 Adaptor_Constructor
Input: *The resulting sets S and T of Algorithm 1*
Initialisation: B_0=$\{\nu \in S\ .\ \nu$ *est bloquant*$\}$.
1. ***For all*** $k \geqslant 0$, *let* $B_{k+1} = B_k \cup PRED^d_T(B_k)$;
2. *Back to the step* 1 ***while*** $B_k \subseteq B_{k+1}$;
Output: *The adaptor Ad of* A_1 *and* A_2 *according to* $\Phi(A_1, A_2)$ *tel que*
- $S_{Ad} = \{\ \nu \in S \setminus B_k\}$ *and*
- $\delta_{Ad} = \{(\nu, a, \nu') \in S_{Ad} \times \Sigma_{Ad} \times S_{Ad}$ *such that* $\nu,\ \nu' \in T \setminus \{(\mu, b, \mu') \in T \mid \mu, \mu' \in S \setminus B_k\ \}\}$.

[4]For a state s, we define by $deg^o_T(s)$ the number of transitions which have s as origin in a set of transitions T.

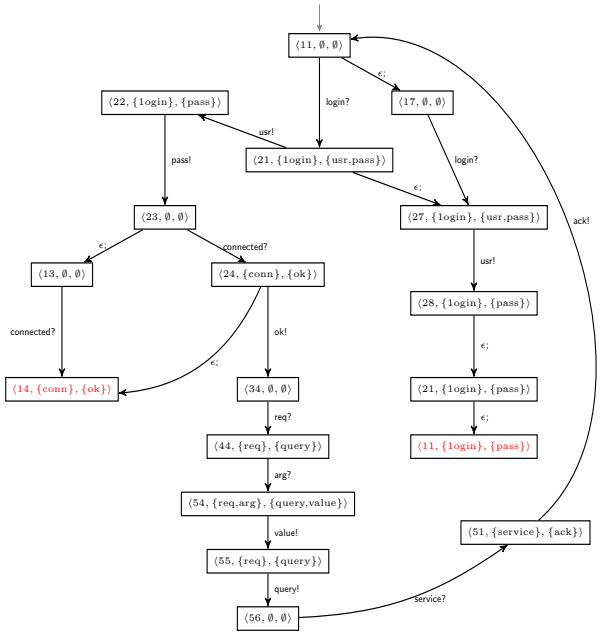

Figure 2. The result of Algorithm 1 for the client/server system of Example 1

By applying Algorithm 2 to the transition system presented in Figure 2, we obtain our desired adaptor shown in Figure 1. It performs a loop back from the blocking states $\langle 11, \{1login!\}, \{pass?\}\rangle$ and $\langle 14, \{connection!\}, \{ok?\}\rangle$ until states which have more than one outgoing transitions. Given two interface automata A_1 and A_2, Algorithm 1 has the complexity $\mathbf{O}(|S_{A_1} \times S_{A_2}|.(|\delta_{A_1}| + |\delta_{A_2}|))$. The complexity of Algorithm 2 is in linear time on the number of states in S.

6 Related Works

Many papers about component adaptation are based on the work presented in [15], where the authors proposed an interesting approach, based on finite state machine, to adapt

components specified by interfaces describing component protocol and action signatures. This approach deals with one-to-one relations between actions. In [7], 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 [14], the authors have developed the tool PaCoSuite to edit visually components, and to generate adaptors using signature and protocols interfaces. In [12], Schmidt and Reussner presented a particular adaptation approach as a solution to synchronisation problems between concurrent components. The approach addresses for instance situations where one component is accessed simultaneously by two other components. The proposed method was implemented in the CoConut/J tool suite [11], where 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.

Others approaches for the generation of adaptor protocols from component behavioral interfaces and composition contracts based on process algebra were proposed in [3, 6]. In [10], the authors developed a game-theoretical approach to find out whether incompatible component interfaces can be made compatible by inserting a converter between them which satisfies specified requirements. In [2], the authors proposed an approach based on transition systems and message sequence charts, for automatic synthesis of failure-free coordinators (adaptor) for COTS component based systems. In [8], the authors proposed a model of adaptors expressed in the B formal method, allowing to define the interoperability between components.

The approaches described above propose solutions for the component adaptation based on different specification formalisms of component interfaces. In our approach, we propose a solution to adapt particular components that are specified by interface automata. This formalism is more general and based on rich notation, allowing to deal with more complex adaptation scenarios.

7 Conclusion and Future Works

In this paper, we propose a methodology for the automatic development of component adaptors, allowing the elimination of behavioural mismatches between interacting components, described by interface automata. We propose an algorithm that generates automatically the adaptor for two adaptable interface automata according to a fixed mapping. The generated adaptor allows to eliminate mismatches at the signature and the protocol levels. The proposed algorithms were implemented in Java in order to validate them, and we plan to propose a complete tool in the future works. Actually, we are developing a tool that implements a framework checking the compatibility between interface automata at the protocol and the semantic levels [9]. We plan also to implement the proposed adaptation approach in our framework.

References

[1] L. Alfaro and T. A. Henzinger. Interface automata. *ACM Press, 9th Annual Aymposium of FSE (Foundations of Software Engineering)*, pages 109–120, 2001.

[2] M. Autili, P. Inverardi, and M. Tivoli. Automatic adaptor synthesis for protocol transformation. In *proceedings of WCAT'04*, pages 39–46, 2004.

[3] A. Bracciali, A. Brogi, and C. Canal. A formal approach to component adaptation. *Journal of Systems and Software*, 74:45–54, 2005.

[4] C.Canal, J. Murillo, and P. Poizat. Software adaptation. *L'OBJET*, 12(1):9–31, 2006.

[5] G. Heineman. An evaluation of component adaptation techniques. *In the proceeding of ICSE Workshop on CBSE*, 1999.

[6] R. Mateescu, P. Poizat, and G. Salaün. Behavioral adaptation of component compositions based on process algebra encodings. In *ASE '07*, pages 385–388, New York, NY, USA, 2007. ACM.

[7] H. Min, S. Choi, and S. Kim. Using smart connectors to resolve partial matching problems in cots component acquisition. *LNCS, SV, Berlin, Germany*, 3054(40-47), 2004.

[8] I. Mouakher, A. Lanoix, , and J. Souquires. Component adaptation: Specification and verification. *In 11th International Workshop on Component Oriented Programming*, 2006.

[9] S. Mouelhi, S. Chouali, and H. Mountassir. Refinement of interface automata strengthened by action semantics. *Electr. Notes Theor. Comput. Sci.*, 253(1):111–126, 2009.

[10] R. Passerone, L. deAlfaro, T. Henzinger, and A. Sangiovanni-Vincentelli. Convertibility verification and converter synthesis: Two faces of the same coin. In *International Conference on Computer Aided Design ICCAD*, pages 132–139. ACM, 2002.

[11] R. Reussner. Automatic component protocol adaptation with the coconut/j tool suite. *Future Generation Comp. Syst.*, 19(5):627–639, 2003.

[12] H. W. Schmidt and R. H. Reussner. Generating adapters for concurrent component protocol synchronisation. In *FMOODS '02*, pages 213–229, Deventer, The Netherlands, The Netherlands, 2002. Kluwer, B.V.

[13] C. Szyperski. Component software: Beyond object oriented programming. *Addison Wesley*, 1999.

[14] W. Vanderperren and B. Wydaeghe. Towards a new component composition process. *In the proceeding of the ECBS 2001 Int Conf, Washington, USA*, pages 322–331, 2001.

[15] D. M. Yellin and R. E. Strom. Protocol specifications and component adaptors. *ACM Trans. Program. Lang. Syst.*, 19(2):292–333, 1997.