Reusing and Adapting Components using atomic and non-atomic Strong Synchronisations

D. Dahmani[1], M.C. Boukala[1], and H. Montassir[2]

[1] MOVEP, USTHB, Algiers.
dzaoueche,mboukala@usthb.dz,
[2] LIFC, Comp. Sci. Dept, Franche-Comté University
hmountassir@lifc.univ-fcomte.fr

Abstract. Composition of heterogeneous software components is required in many domains to build complex systems. However, such compositions raise mismatches between components. Software adaptation aims at generating adaptors to correct mismatches between components to be composed. In this paper, we propose a formal approach which relies on mapping rules to generate automatically adaptors and check compatibilities of components. Our solution allows both atomic and non-atomic synchronisations. We adopt a strong synchronisation semantic which comes naturally with the semantic of the synchronous product of automata.

Keywords: Interface automata, components reuse, components adaptation, atomic and non-atomic synchronisation.

1 Introduction

Component-based development aims at facilitating the construction of very complex and huge applications by supporting the composition of simple building existing modules, called components. The assembly of components offers a great potential for reducing cost and time to build complex software systems and improving system maintainability and flexibility. The reuse of a component and substitution of an old component by a new one are very promising solution [9, 10] .

A component is a software unit characterised by an interface which describes the services offered or required by the component, without showing its implementation. In other terms, only information given by a component interface are visible for the other components. Moreover, interfaces may describe component information at signature level (method names and their types), behaviour or protocol (scheduling of method calls) and method semantics.

A software component is generally developed independently and is subject to assembly with other components, which have been designed separately, to create a system. Normally glue code is written to realise such assembly. Unfortunately, components can be incompatible and cannot work together. Two components are

incompatible if some services requested by one component cannot be provided by the other [1, 5]. The pessimistic approach considers two components compatible if they can always work together. Whereas, in the optimistic approach two components are compatible if they can be used together in at least one design [1].
Incompatibilities are identified: (i) at signature level coming from different names of methods, types or parameters and (ii) at behaviour or protocol level as incompatible orderings of messages [4]. To correct such incompatibilities, adaptors are used as components that can be plugged between the mismatched components, instead of re-designing, re-implementing and re-testing new component codes. Some works are in favor of automated generation adaptor components. For instance, in [3] an automated generation of adaptors is proposed capable of solving behaviour mismatches. Given two processes whose interactions may lock, an adaptation process is built allowing a successful cooperation between the former processes. This approach assumes that there is no mismatch at the interface level. Other approaches rely on matching rules provided by the developer to generate automatically adaptors for components with signature mismatches. Other works focus rather on semi-automated adaptation of mismatched components. Some of them deal with both behaviour and interface levels. In [8], annotations of component or services interfaces using the Web Ontology language (OWL) [2] in relation to some domain concepts thereby allowing services to be semantically matched, based on their ontological annotations. By inferences on this meta-data, services that are syntactically different but semantically equivalent may be autonomously adapted and substituted. The limits of such works are mainly due to the limited availability of ontologies.

In [11], authors propose techniques for the identification and resolution of mismatches between services. First signature matchings based on XML schema matching are used for the resolution of signature mismatches. Furthermore, unspecified receptions are automatically dealt, contrary to deadlock interactions between services that are semi-automatically tackled. In fact, deadlock scenarios are generated, helping the developer to specify adaptation contracts. This approach has been improved in [14]: (i) to extend matching schemas, particularly to support one-to-many correspondence between operations. (ii) to support the protocol-level mismatch more efficient.

In [4], an interactive approach to support the contract design process is given. A graphical environment allows to define port bindings and suggest some port connections to the designer based on interface compatibility measures. Compositional and hierarchical techniques are proposed to facilitate the specification of adaptation contracts by building them incrementally. Validation and verification techniques are also proposed to check that the contract will make the involved services work as expected.

Some of the mentioned works are based on formal methods such as interface automata, logic formula and Petri nets which give formal description to software interface and behaviour [5, 7].

In [6], an algorithm for adaptor construction based on interface automata is proposed. Such adaptors operate at signature level and rely on mapping rules. The adaptors are represented by interface automata which aim at converting data between components according to mapping rules. However, the proposed approach yields problematic interactions between adapted components. Our work is a continuation of [6] aiming at overcoming such interactions and offering a clear semantics of the adaptor specifications. Our approach extends the synchronous product of interface automata presented in [1] to allow both atomic and non-atomic synchronisation. However, to accurately preserve the semantics of the synchronous product of automata, we opt for a *strong* synchronisation semantics, i.e. *almost one non-atomic synchronisation* between components.

This paper contains five sections. Section 2 is consecrated to describe interface automata. The concept of mapping rules is given in section 3. In section 4, we describe our component adaptation approach. Finally, we conclude and present some perspectives.

2 Interface Automata

Interface automata are introduced by L.Alfaro and T.Henzinger [1], to model component interfaces. Input actions of an automaton model offered services, i.e. methods that can be called or messages reception. Whereas output actions are used to model method calls and message transmissions. Internal actions represent hidden actions of the component. Moreover, interface automata interact through the synchronisation of input and output actions, while internal actions of concurrent automata are interleaved asynchronously.

Definition 1 *(Interface automaton)*
An interface automaton $A = \langle S_A, S_A^{init}, \Sigma_A, \tau_A \rangle$ *where :*

1. S_A *is a finite set of states,*
2. $S_A^{init} \subseteq S_A$ *is a set of initial states. If* $S_A^{init} = \emptyset$ *then* A *is empty,*
3. $\Sigma_A = \Sigma_A^O \cup \Sigma_A^I \cup \Sigma_A^H$ *a disjoint union of output, input and internal actions,*
4. $\tau_A \subseteq S_A \times \Sigma_A \times S_A$.

The input or output actions of automaton A are called external actions denoted by $\Sigma_A^{ext} = \Sigma_A^O \cup \Sigma_A^I$. A is closed if it has only internal actions, that is $\Sigma_A^{ext} = \emptyset$; otherwise we say that A is open. Input, output and internal actions are respectively labelled by the symbols "?", "!" and ";". In the sequel, we use δ and $\overline{\delta}$ to prefix actions, with $\delta \in \{!, ?\}$ and $\overline{\delta} \in \{!, ?\} \setminus \{\delta\}$. An action $a \in \Sigma_A$ is enabled at a state $s \in S_A$ if there is a step $(s,a,s') \in \tau_A$ for some $s' \in S_A$.

Example 1

Fig. 1 depicts a modelling of remote accesses to a data base. This example will be used throughout this paper. The system contains two components *Client* and *Server* which have been designed separately. On the one hand, *Client* issues an authentication message (!*login*). If *Client* is not authenticated by *Server* (?*nAck* and ?*errN*), it exits. Otherwise, *Client* loops on sending read or update requests. When *Client* sends a read request (!*req*) followed by its arguments (!*arg*), it waits the result (?*data*). An update request is an *atomic* action (!*update*). At any moment, *Client* can exit (!*logout*). *Client* may receive a ?*halt* signal from the environment, then it exits.
On the other hand, when *Server* receives a login message (?*login*), it either accepts the client access request (!*ok*) or denies it (!*nOk*). Afterwards, *Server* becomes ready to receive a read or update requests. If it receives a read request (?*query*), it performs a local action (;*rdDB*) and sends the appropriate data (!*data*). *Server* can accept a *logout* message (?*logout*).

Fig. 1.a depicts interface automaton *Server*. It is composed of five states ($s_0, \ldots s_4$), with state s_0 being initial, and eight steps, for instance (s_0, ?*login*,s_1). The sets of input, output and internal actions are given below:
- Σ^O_{Server} =$\{ok, nOk, data\}$,
- Σ^I_{Server} = $\{login, logout, query, update\}$,
- Σ^H_{Server}= $\{rdDB\}$.

2.1 Composition of interface automata

Let A_1 and A_2 two automata. An input action of one may coincide with a corresponding output action of the other. Such an action is called a shared action. We define the set $\Sigma_{shr(A_1,A_2)} = \{a \mid \delta a \in \Sigma_{A_1} \wedge \overline{\delta} a \in \Sigma_{A_2}\}$, e.g. set $\Sigma_{shr(Client,Server)} = \{login, logout, update, data\}$.

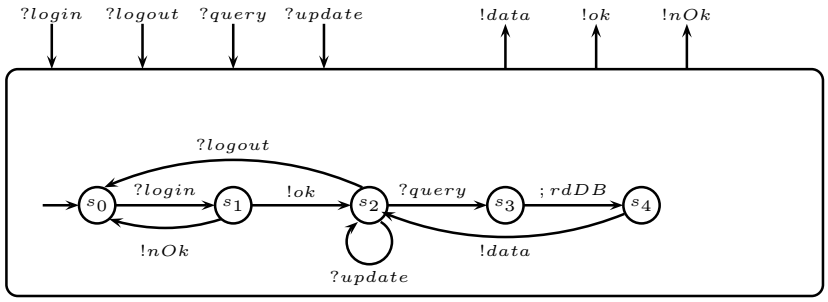
The composition of two interface automata is defined only if their actions are disjoint, except shared input and output ones. The two automata will synchronize on shared actions, and asynchronously interleave all other actions [1].

Definition 2 *(Composable automata)*
Two interface automata A_1 and A_2 are composable iff

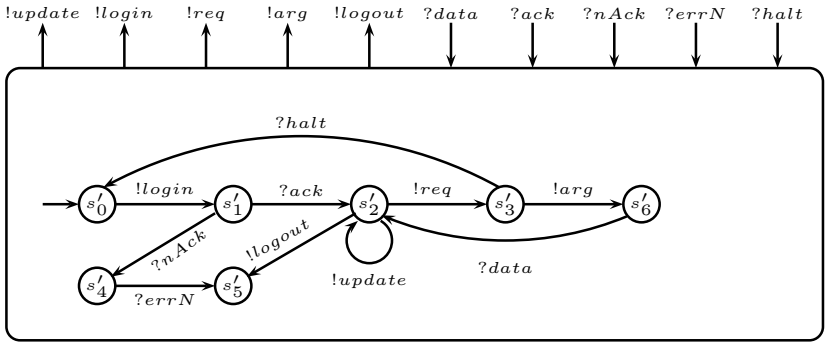
$$(\Sigma^H_{A_1} \cap \Sigma_{A_2} = \emptyset) \wedge (\Sigma^H_{A_2} \cap \Sigma_{A_1} = \emptyset) \wedge (\Sigma^I_{A_1} \cap \Sigma^I_{A_2} = \emptyset) \wedge (\Sigma^O_{A_1} \cap \Sigma^O_{A_2} = \emptyset)$$

Definition 3 *(Synchronous product)*
If A_1 and A_2 are composable interface automata, their product $A_1 \otimes A_2$ is the interface automaton defined by:

1. $S_{A_1 \otimes A_2} = S_{A_1} \times S_{A_2}$,
2. $S^{int}_{A_1 \otimes A_2} = S^{int}_{A_1} \times S^{int}_{A_2}$,
3. $\Sigma^H_{A_1 \otimes A_2}$=$(\Sigma^H_{A_2} \cup \Sigma^H_{A_1}) \cup \Sigma_{shr(A_1,A_2)}$,
4. $\Sigma^I_{A_1 \otimes A_2}$=$(\Sigma^I_{A_1} \cup \Sigma^I_{A_2}) \setminus \Sigma_{shr(A_1,A_2)}$,
5. $\Sigma^O_{A_1 \otimes A_2}$=$(\Sigma^O_{A_1} \cup \Sigma^O_{A_2}) \setminus \Sigma_{shr(A_1,A_2)}$,



(a) Server



(b) Client

Fig. 1: *Server* and *Client* interface automata

6. $\tau_{A_1 \otimes A_2} = \{(v,u),a,(v',u) \mid (v,a,v') \in \tau_{A_1} \wedge a \notin \Sigma_{shr(A_1,A_2)} \wedge u \in S_{A_2}\}$
$\cup \{(v,u),a,(v,u') \mid (u,a,u') \in \tau_{A_2} \wedge a \notin \Sigma_{shr(A_1,A_2)} \wedge v \in S_{A_1}\}$
$\cup \{(v,u),a,(v',u') \mid (v,a,v') \in \tau_{A_1} \wedge (u,a,u') \in \tau_{A_2} \wedge a \in \Sigma_{shr(A_1,A_2)}\}$.

Each state of the product consists of a state of A_1 together with a state of A_2 (1). Initial states of the product $A_1 \otimes A_2$ are also initial states in A_1 and A_2. A shared action of $\Sigma_{shr(A_1,A_2)}$ is *internal* for $A_1 \otimes A_2$. Moreover, any internal action of A_1 or A_2 is also internal for $A_1 \otimes A_2$ (3). The input (resp. output) actions of one automaton which are not shared with the other are input (resp. output) ones for $A_1 \otimes A_2$ (4, 5). Each step of the product is either a joint shared action step or a non shared action step in A_1 or A_2 (6).

In the product $A_1 \otimes A_2$, one of the automata may produce an output action that is an input action of the other automaton, but is not accepted. A state of $A_1 \otimes A_2$ where this occurs is called an illegal state of the product. When $A_1 \otimes A_2$ contains illegal states, A_1 and A_2 can't be composed in the pessimistic approach. In the optimistic approach A_1 and A_2 can be composed provided that there is an adequate environment which avoids illegal states. For more details, we refer the reader to [1].

Incompatibilities may also come from the nature of exchanged information. Synchronous product, as defined in this section, does not deal with such incompatibilities. As mentioned in the introduction, adaptors are required to convert the exchanged information causing mismatches. In particular, mapping rules are used to adapt exchanged action names between the components. Such rules may be given by designer. For more details, we refer reader to [12].

3 Mapping Rules for Incompatible Components

A mapping rule establishes correspondence between some actions of A_1 and A_2. Each mapping rule of A_1 and A_2 associates an action of A_1 with more actions of A_2 (one-for-more) or vice versa (more-for-one).

Definition 4 *(Mapping rule)*
A mapping rule of two composable interface automata A_1 and A_2 is a couple $(L_1,L_2) \in (2^{\Sigma^{ext}_{A_1}} \times 2^{\Sigma^{ext}_{A_2}})$ *such that* $(L_1 \cup L_2) \cap \Sigma_{shr(A_1,A_2)} = \emptyset$ *and if* $|L_1| > 1$ *(resp.* $|L_2| > 1$*) then* $|L_2| = 1$ *(resp.* $|L_1| = 1$*).*

A mapping $\Phi(A_1,A_2)$ of two composable interface automata A_1 and A_2 is a set of mapping rules associated with A_1 and A_2.
We denote by $\Sigma_{\Phi(A_1,A_2)}$ the set $\{a \in \Sigma^{ext}_{A_1} \cup \Sigma^{ext}_{A_2} | \exists \alpha \in \Phi(A_1,A_2)$ s.t $a \in \Pi_1(\alpha) \cup \Pi_2(\alpha)\}$, with $\Pi_1(\langle L_1,L_2\rangle) = L_1$ and $\Pi_2(\langle L_1,L_2\rangle) = L_2$ are respectively the projection on the first element and the second one of the couple $\langle L_1,L_2\rangle$. Observe that each action of $\Sigma_{\Phi(A_1,A_2)}$ is a source of mismatch situation between A_1 and A_2.

Example 2

Consider again the components of example 1. In *Server* a read request is viewed into one part ($?query$), whereas it is structured as two parts ($!req, !arg$) in *Client*. A mapping rule is necessarily to map $\{!req, !arg\}$ to $\{?query\}$. The sets of mapping rules between *Client* and *Server* $\Phi_{(Client,Server)}$ and $\Sigma_{\Phi(Client,Server)}$ are defined as follows:

1. $\Phi_{(Client,Server)} =$
   { ({?nAck, ?errN}, {!nOk})
   ($\{?ack\}$, $\{!ok\}$),
   ($\{!req$, $!arg\}$, $\{?query\}$)}

2. $\Sigma_{\Phi(Client,Server)} = \{!req, !arg, ?query, ?ack, !ok, ?nAck, !nOk, ?errN\}$

4 Towards Component Adaptation

In $A_1 \otimes A_2$, the actions of $\Sigma_{\Phi(A_1,A_2)}$ are interleaved asynchronously since they are named differently in A_1 and A_2. As mentioned, an adaptor component, must be defined. Such an adaptor, denoted by Ad, is mainly based on the set $\Phi(A_1, A_2)$ and is a mediator between A_1 and A_2.

Definition 5 *(Adaptation of A_1 and A_2)*
The automata A_1 and A_2 are adaptable according to $\Phi(A_1,A_2)$ if (i) A_1 and A_2 are composable, (ii) $\Phi(A_1, A_2)$ is not empty and (iii) there is a non empty automaton adaptor Ad.

Adapter Ad receives the output actions specified in $\Sigma_{\Phi(A_1,A_2)}$ from one automaton and sends the corresponding input actions to the other. Thus, the adaptation of A_1 and A_2 yields to atomic and non-atomic synchronisation. The former use shared actions and hence are classical synchronisation; the second ones use mismatch actions and need several steps. Thus, synchronisation actions may be mixed. In this work, we deal with strong synchronisation between automata, where a non-atomic synchronisation occurrence freezes temporally any other synchronization between A_1 and A_2. However, internal actions of A_1 and A_2 and synchronization with other components (distinct of A_1 and A_2) are allowed.

4.1 Description of the adapter construction

The actions of $\Sigma_{\Phi(A_1,A_2)}$ transit by adapter Ad. The *input* actions of $\Sigma_{\Phi(A_1,A_2)}$ are *outputs* for Ad and vice versa, e.g $!req \in \Sigma_{\Phi(A_1,A_2)}$ and $?req \in \Sigma^I_{Ad}$. Adaptor has no inherent activity. However, it has invisible actions, represented by ϵ, which are used for implementation requirement.

Notations and Conventions
In the sequel, $\Sigma_{syn(A_1,A_2)}$, denotes the set of actions involved in actomic or non-atomic synchronisation between A_1 and A_2 (i.e. $\Sigma_{shr(A_1,A_2)} \cup \Sigma_{\Phi(A_1,A_2)}$). Let α be a mapping rule of $\Phi(A_1, A_2)$, an α-synchronisation of A_1 and A_2 is a non-atomic synchronisation throw adapter Ad, involving only actions of α.

A state s is a tuple $\langle s_1, s_2, R, O, I\rangle$ where s_1 is a state of A_1 and s_2 of A_2. R is the current mismatch rule. When $R = \alpha \in \Phi(A_1,A_2)$, then state s is a progression of an α-synchronisation occurrence. Furthermore, (i) O gives the output actions of α already encountered in states preceding s, and (ii) I the input actions of α expected in states following s. When $R = null$, sets O and I are empty, meaning that there is no non-atomic synchronisation in progress.

Definition 6 *(Adapter automata construction)*
Let two composable automata A_1, A_2 and a non-empty set of mapping $\Phi(A_1,A_2)$, an adapter of A_1 and A_2 according to $\Phi(A_1, A_2)$ is an interface automaton defined by:
Σ^I_{Ad}=$\{a \in \Sigma^O_{A_1} \cup \Sigma^O_{A_2} | a \in \Sigma_{\Phi}(A_1,A_2)\}$,
Σ^O_{Ad}=$\{a \in \Sigma^I_{A_1} \cup \Sigma^I_{A_2} | a \in \Sigma_{\Phi}(A_1,A_2)\}$,
Σ^H_{Ad}=$\{\epsilon\}$,
$S_{A_d} = S_{A_1} \times S_{A_2} \times (\Phi(A_1, A_2) \cup \{null\}) \times 2^{\Sigma^I_{Ad}} \times 2^{\Sigma^O_{Ad}}$,
$S^{int}_{Ad} = S^{int}_{A_1} \times S^{int}_{A_2} \times \{null\} \times \emptyset \times \emptyset$,

The set τ_{Ad} are defined as follows:

1. ***If*** $s_1 \xrightarrow{\delta a} s'_1 \in \tau_{A_1} \wedge a \in \Sigma_{A_1} \setminus \Sigma_{syn(A_1,A_2)}$
then $(s_1, s_2, R, O, I) \xrightarrow{\epsilon} (s'_1, s_2, R, O, I) \in \tau_{Ad}$.

2. ***If*** $s_1 \xrightarrow{\delta a} s'_1 \in \tau_{A_1} \wedge s_2 \xrightarrow{\overline{\delta a}} s'_2 \in \tau_{A_2} \wedge\ R = null \wedge a \in \Sigma_{shr}(A_1, A_2)$
then $(s_1, s_2, R, O, I) \xrightarrow{\epsilon} (s'_1, s'_2, R, O, I) \in \tau_{Ad}$.

3. ***If*** $s_1 \xrightarrow{!a} s'_1 \in \tau_{A_1} \wedge \exists \alpha \in \Phi(A_1,A_2)$ *s.t.* $a \in \Pi_1(\alpha) \wedge a \notin O \wedge (R = null \vee R = \alpha)$
then
- $R' \leftarrow \alpha$, $O' \leftarrow O \cup \{a\}$,
- ***If*** $O' = \Pi_1(\alpha)$ ***then*** $I' = \Pi_2(\alpha)$ ***else*** $I' = I$.
- $(s_1, s_2, R, O, I) \xrightarrow{?a} (s'_1, s_2, R', O', I') \in \tau_{Ad}$

4. ***If*** $s_1 \xrightarrow{?a} s'_1 \in \tau_{A_1} \wedge \exists \alpha \in \Phi(A_1,A_2)$ *s.t.* $a \in \Pi_1(\alpha) \wedge a \in I \wedge R = \alpha$
then
- $I' \leftarrow I \setminus \{a\}$,
- ***If*** $I' = \emptyset$ ***then*** $O' \leftarrow \emptyset$, $R' \leftarrow null$ ***else*** $R' \leftarrow \alpha$
- $(s_1, s_2, R, O, I) \xrightarrow{!a} (s'_1, s_2, R', O', I') \in \tau_{Ad}$

5. Adapt points 1, 3 and 4 for the automaton A_2.

A step $s \xrightarrow{\epsilon} s' \in \tau_{Ad}$ models an invisible action (ϵ) and is associated with a step of A_1 or A_2, involving some action a which is not handled by adaptor Ad. Such an action may be internal for one of the automata A_1 or A_2, or an external action of A_1 (resp. A_2) but is not shared with A_2 (resp. A_1) (point 1) or a shared synchronization action between A_1 and A_2 (point 2).

Let a be an action handled by adaptor Ad, then a belongs to some rule of $\Phi(A_1,A_2)$, say α. The visible steps of Ad are built as follows:

- An input step $s \xrightarrow{?a} s' \in \tau_{Ad}$ represents an output step of a in A_1 or A_2. When R is *null* in state s, the step of Ad initiates a new occurrence of an α-synchronisation between A_1 and A_2 through Ad. When R is set to α in state s, the step of Ad is a progression of an α-synchronisation. In this case, action a must not have already been encountered for the current α-synchronisation occurrence, that means $a \notin O$. In state s', a is considered as encountered ($a \in O'$). When all output actions of α have been encountered at state s', the input actions of α will be expected in future states, then I' is set to the input actions of α.

- An output step $s \xrightarrow{!a} s' \in \tau_{Ad}$ represents an input step of a in A_1 or A_2. The step of Ad is necessarily a progression of an α-synchronisation occurrence.

Moreover, action a is expected at state s, i.e. $a \in I$ in s. If a is the last expected action, this step achieves the current α-synchronisation occurrence, therefore, R' is set to *null*, O' and I' to empty set.

Fig. 2 gives the adaptor of *Client* and *Server* according to the mismatch rules given in example 2. The grey state is **terminal** with fields R, O and I not empty. Such a state indicates a mismatch situation. Valid Adapter automaton is obtained by removing such states, and repeat removing until no state have its fields R, O or I not empty.

4.2 Adapter properties

By construction, the following properties hold:

1. (*No Mix*)
   $\forall a \in \Sigma_{shr(A_1,A_2)}$, $\forall\ s = \langle s_1, s_2, R, O, I\rangle$ a state of Ad,
   $\exists s_1 \xrightarrow{\delta a} s_1' \in \tau_{A_1}$ and $\exists s_2 \xrightarrow{\overline{\delta} a} s_2' \in \tau_{A_2} \Rightarrow R = null$, $O = \emptyset$ and $I = \emptyset$.

2. (*Coherence*)
   $\forall\ \sigma$ a sequence of Ad such that each of its state has its third field non *null*, except the states of the extremities, the following points hold:
   - The intermediate states of σ are attached to a same rule, say α.
   - Each action of α appears once and only once in σ; the output actions of α (i.e. which are necessarily input actions of Ad) appear before its input ones.
   - ϵ actions may appear in σ. They come from actions which are not synchronisation between A_1 and A_2.
3. (*Illegal state*)
   Any *terminal* state $s = \langle s_1, s_2, R, O, I\rangle$ of Ad such that O or I are not empty indicates a **failure** of a non-atomic synchronization.

Properties 1 and 2 ensure a strong synchronization between actions of A_1 and A_2. Whereas, property 3 allows to detect incompatibilities in the composition of A_1 and A_2.

Fig. 3 highlights the composition of *Client*, *Server* and their adapter *Ad*. The grey state is illegal. However, it is possible to avoid such a state (in the optimist approach) in an environment where *halt* signal is never sent to *Client*.

5 Conclusion

Software adaptation is widely used for adapting incompatible components, viewed as black boxes. In this paper, we presented simple and clear automata construction for software adaptation based on mapping rules. These later are used to

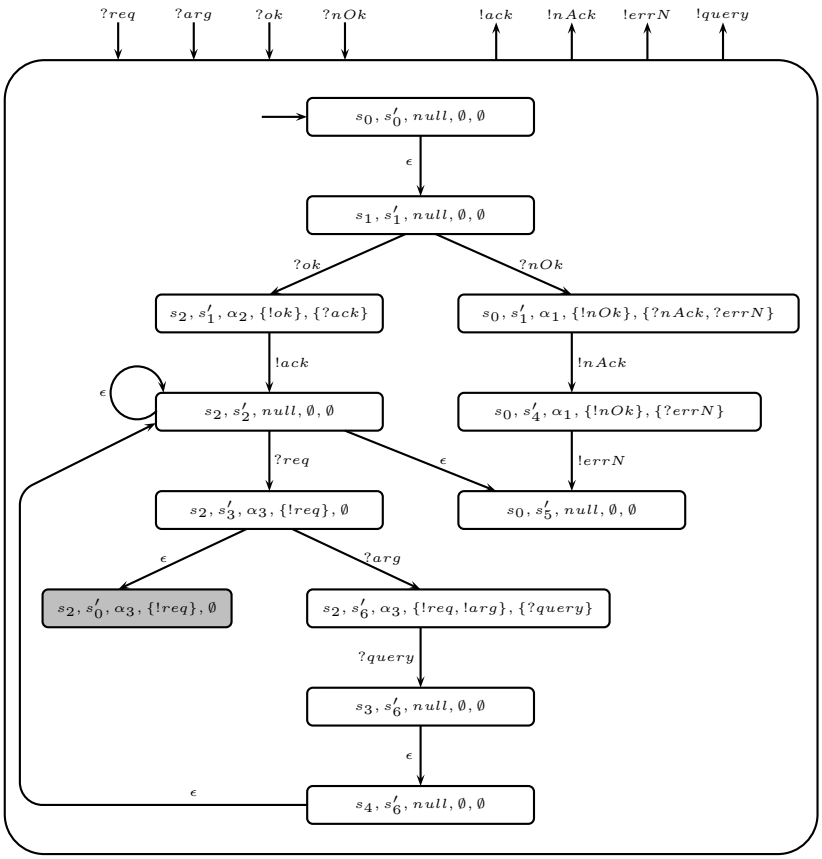
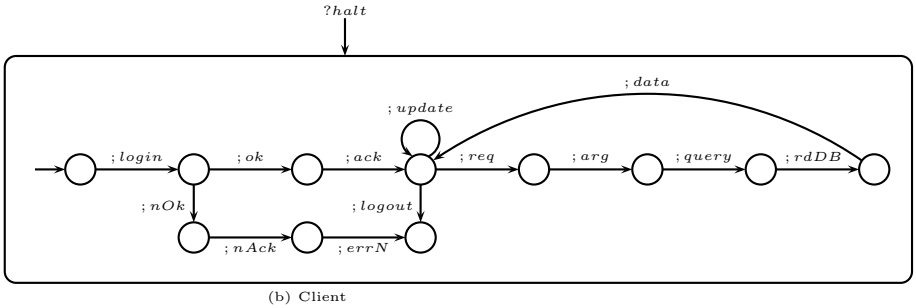


Fig. 2: *Server* and *Client* Adapter



(b) Client

Fig. 3: *Client* $\otimes$ *Adapter* $\otimes$ *Server*

express correspondence between mismatch actions of components and are behind non-atomic synchronisation. In our solution, synchronisations between same components can not be interfered. We focus our future work in two directions.

To validate and illustrate our approach, before implementing the algorithm to generate the adapter, we plane to use Ptolemy [13] tool to detect incompatibilities. The second direction consists to extend component interfaces to take into account adaptation of components with temporal constraints and verification of temporal properties of the system.

References

1. L. Alfaro and T.A. Henzinger. Interface automata. In *Proceedings of the Ninth Annual Symposium on Foundations of Software Engineering (FSE), ACM*, pages 109–120. Press, 2001.
2. Grigoris Antoniou, , Grigoris Antoniou, Grigoris Antoniou, Frank Van Harmelen, and Frank Van Harmelen. Web ontology language: Owl. In *Handbook on Ontologies in Information Systems*, pages 67–92. Springer, 2003.
3. Antonio Brogi and Razvan Popescu. Automated generation of bpel adapters. In Francisca Losavio, Guilherme Horta Travassos, Vicente Pelechano, Isabel Daz, and Alfredo Matteo, editors, *CIbSE*, pages 387–400, 2007.
4. Javier Camára, Gwen Salaün, Carlos Canal, and Meriem Ouederni. Interactive specification and verification of behavioral adaptation contracts. *Information and Software Technology*, 54(7):701–723, 2012.
5. C. Canal, P. Poizat, and G. Salaun. Model-based adaptation of behavioral mismatching components. *IEEE Transactions on Software Engineering*, 34(4):546–563, 2008.
6. S. Chouali, S. Mouelhi, and H. Mountassir. Adapting components behaviours using interface automata. In *SEAA'10, 36th Euromicro Conference on Software Engineering and Advanced Applications*, pages 119–122, Lille, France, September 2010. IEEE Computer Society Press.
7. S. Chouali, S. Mouelhi, and H. Mountassir. Adapting components using interface automata strengthened by action semantics. In *FoVeoos 2010, int. conf. on Formal Verification of Object-oriented software*, pages 7–21, Paris, France, June 2010.
8. Jeffrey Hau, William Lee, and Steven Newhouse. Autonomic service adaptation in iceni using ontological annotation. In *Proceedings of the 4th International Workshop on Grid Computing*, GRID '03, pages 10–, Washington, DC, USA, 2003. IEEE Computer Society.
9. C.W. Krueger. Software reuse. *ACM Comput. Surv.*, 24(2):131–183, June 1992.
10. L. Kung-Kiu and W. Zheng. Software component models. *IEEE Transactions on Software Engineering*, 33(10):709–724, 2007.
11. Hamid Reza Motahari Nezhad, Boualem Benatallah, Axel Martens, Francisco Curbera, and Fabio Casati. Semi-automated adaptation of service interactions. In *Proceedings of the 16th International Conference on World Wide Web*, WWW '07, pages 993–1002, New York, NY, USA, 2007. ACM.
12. H.R. Motahari Nezhad, B. Benatallah, A. Martens, F. Curbera, and F. Casati. Semi-automated adaptation of service interactions. In *WWW*, pages 993–1002, 2007.
13. Claudius Ptolemaeus, editor. *System Design, Modeling, and Simulation using Ptolemy II*. Ptolemy.org, 2014.
14. Gwen Salaün, Lucas Bordeaux, and Marco Schaerf. Describing and reasoning on web services using process algebra. In *Proceedings of the IEEE International Conference on Web Services*, ICWS '04, pages 43–, Washington, DC, USA, 2004. IEEE Computer Society.