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

D. Dahmani¹, M.C. Boukala¹ and H. Montassir²

(1) MOVEP, Dépt Informatique, USTHB
(2) LIFC, Comp. Sci. Dept, Franche-Comté University
Component-based development aims at:

- Reusing existing components.
- Facilitating the construction of very complex and huge applications.
- Reducing cost and time.
- Improving system maintainability and flexibility.
Component Interface

- The specification of the offered or required services (operations)

Component Protocol

- Specify the ordering constraints on the exchange of messages
Motivation

Components Interoperability

- The typical approach to make interoperation of heterogeneous software components possible consists of using adapters.
- An adapter is used as a component that can be plugged between the heterogeneous components,

\[ \text{Diagram:}\]

- `?b` to `!a` and `?c` to `!e`
Components Interoperability

- The typical approach to make interoperation of heterogeneous software components possible consists of using adapters.
- An adapter is used as a component that can be plugged between the heterogeneous components,
Outline

Overview

Interface Automata

Mapping Rules for Incompatible Components

Description of the adapter construction

Conclusion
Overview

Interface Automata

Mapping Rules for Incompatible Components

Description of the adapter construction

Conclusion
Overview

Interface Automata

Mapping Rules for Incompatible Components

Description of the adapter construction

Conclusion
Outline

Overview

Interface Automata

Mapping Rules for Incompatible Components

Description of the adapter construction

Conclusion
Outline

Overview

Interface Automata

Mapping Rules for Incompatible Components

Description of the adapter construction

Conclusion
A classification of possible interface mismatches

- Different names,

- Different types,

- Different structure,
A classification of possible protocol mismatches

- **Deadlock,**

  ![Diagram for deadlock]

- **Message missing,**

  ![Diagram for message missing]

- **Message ordering,**

  ![Diagram for message ordering]
Related Works

Automated generation of adapters

- Assume no interface mismatch [BP07] or matching rules given by the developer ([CMM10])
- Propose automatic building of an adaptation process for BPEL processes whose interaction may lock ([BP07])

Semi-automated adaptation

- Identification and resolution of mismatches at interface-level by using:
  - Web Ontology Language OWL and inference rules ([AA+03])
  - XML schema ([MNBM+07])
- Identification of mismatches at protocol-level ([MNBM+07])
  - Automatic management of unspecified reception.
  - Semi-automatic management of deadlock interactions and help for contract construction.
Definition
An interface automaton $A = \langle S_A, S_{A}^{\text{init}}, \Sigma_A, \tau_A \rangle$ where:

1. $S_A$ is a finite set of states,
2. $S_{A}^{\text{init}} \subseteq S_A$ is a set of initial states. If $S_{A}^{\text{init}} = \emptyset$ 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. $\tau_A \subseteq S_A \times \Sigma_A \times S_A$. 
Interface Automata (Example)

Example

(a) Server

(b) Client
Example

\[ \Sigma^O_{\text{Server}} = \{ \text{ok}, \text{nOk}, \text{data} \} \]

\[ \Sigma^I_{\text{Server}} = \{ \text{login}, \text{logout}, \text{query}, \text{update} \} \]

\[ \Sigma^H_{\text{Server}} = \{ \text{rdDB} \} \]
Composition of interface automata

Let $A_1$ and $A_2$ two interface 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 $\Sigma_{shr}(A_1, A_2) = \{ a \mid \delta a \in \Sigma_{A_1} \land \overline{\delta a} \in \Sigma_{A_2} \}$, e.g. set $\Sigma_{shr}(\text{Client}, \text{Server}) = \{ \text{login, logout, update, data} \}$. 

- Actions are disjoint, except shared input and output ones.

- The two automata will synchronize on shared actions, and asynchronously interleave all other actions.

**Definition**

Two interface automata $A_1$ and $A_2$ are composable iff

$$(\Sigma^H_{A_1} \cap \Sigma_{A_2} = \emptyset) \land (\Sigma^H_{A_2} \cap \Sigma_{A_1} = \emptyset) \land (\Sigma^I_{A_1} \cap \Sigma^I_{A_2} = \emptyset) \land (\Sigma^O_{A_1} \cap \Sigma^O_{A_2} = \emptyset)$$
Synchronous product of interface automata

Definition

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. $S_{A_1 \otimes A_2}^{int} = S_{A_1}^{int} \times S_{A_2}^{int}$,
3. $\Sigma_{A_1 \otimes A_2}^H = (\Sigma_{A_2}^H \cup \Sigma_{A_1}^H) \cup \Sigma_{shr(A_1,A_2)}$,
4. $\Sigma_{A_1 \otimes A_2}^I = (\Sigma_{A_1}^I \cup \Sigma_{A_2}^I) \setminus \Sigma_{shr(A_1,A_2)}$,
5. $\Sigma_{A_1 \otimes A_2}^O = (\Sigma_{A_1}^O \cup \Sigma_{A_2}^O) \setminus \Sigma_{shr(A_1,A_2)}$,
6. $\tau_{A_1 \otimes A_2} = \{(v, u), a, (v', u) | (v, a, v') \in \tau_{A_1} \land a \notin \Sigma_{shr(A_1,A_2)} \land u \in S_{A_2}\}$
   $\cup \{(v, u), a, (v, u') | (v, a, u') \in \tau_{A_2} \land a \notin \Sigma_{shr(A_1,A_2)} \land v \in S_{A_1}\}$
   $\cup \{(v, u), a, (v', u') | (v, a, v') \in \tau_{A_1} \land (u, a, u') \in \tau_{A_2} \land a \in \Sigma_{shr(A_1,A_2)}\}$. 


Incompatible interface automata

- In the product $A_1 \otimes A_2$, one of the automata may produce an output action that is not accepted by the other. A state of $A_1 \otimes A_2$ where this occurs is called an illegal state.
- 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.
Incompatible interface automata

Mismatch situations between component interfaces may be caused by:
- message (services) names that do not correspond,
- an ordering of messages (services) which is not compatible,
- some messages in one component that have no counterpart (one-to-zero),
- some messages match with several messages in another component (one-to-many)

All these cases of behavioural mismatch can be worked out by using Adapters, to convert the exchanged information causing mismatches.

mapping rules are used to adapt exchanged action names between the components.
Mapping Rules for Incompatible Components

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 \textit{Client} and \textit{Server} are defined as follows:

\[
\Phi_{(\text{Client,Server})} = \\
\{ \{ ?\text{nAck}, ?\text{errN} \}, \{ !\text{nOk} \} \}, \{ ?\text{ack} \}, \{ !\text{ok} \} \}, \{ \{ !\text{req}, !\text{arg} \}, \{ ?\text{query} \} \}
\]
Mapping Rules for Incompatible Components

Shared actions: $\Sigma_{shr}(Client,Server) = \{login, logout, update, data\}$.

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

Synchronization actions: $\Sigma_{syn}(Client,Server) = \Sigma_{shr}(Client,Server) \cup \Sigma_{\Phi}(Client,Server)$
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**

A mapping rule of two composable interface automata $A_1$ and $A_2$ is a couple $(L_1, L_2)$ $\in (2^{\Sigma_{A_1}^\text{ext}} \times 2^{\Sigma_{A_2}^\text{ext}})$ 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. $|L_1| = 1$).
Description of the adapter construction

- The adapter is a mediator between $A_1$ and $A_2$ and aims at converting mismatch actions.

- This adapter is mainly based on the set $\Phi(A_1, A_2)$.

- The adapter receives the output mismatch actions from one automaton and sends the corresponding input actions to the other.
Description of the adapter construction

- When it is plugged between $A_1$ and $A_2$, a possible step in one automaton:
  - an internal step,
  - A communication step with the environment,
  - A shared action communication (atomic synchronisation) with the other automaton,
  - A mismatch action communication (non-atomic synchronisation) towards the other automaton, transiting by the adapter

- Whatever its kind, almost one synchronisation between the automata can be on. Therefore, a non-atomic synchronisation occurrence freezes temporally any other synchronisation between them.
The protocol behaviour of the Adapter is represented by an interface automaton.

A state of the Adapter is a tuple $s = < s_1, s_2, R, O, I >$ such that:

- $s_1$ and $s_2$ are states of $A_1$ and $A_2$, respectively.
- $R$, $O$, $I$ are used for non-atomic synchronisation implementation requirements.
- During the construction of the adapter $R$ is set either to a matching rule $\alpha$ or to null:
  - Whenever $R = \alpha$, $O$ contains the output actions already encountered, $I$ contains the input action expected in the following states.
  - When $R = \text{null}$, $O$ and $I$ are empty.
Definition
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:

$$\Sigma^I_{Ad} = \{a \in \Sigma^O_{A_1} \cup \Sigma^O_{A_2} | a \in \Sigma_\Phi(A_1, A_2)\},$$

$$\Sigma^O_{Ad} = \{a \in \Sigma^I_{A_1} \cup \Sigma^I_{A_2} | a \in \Sigma_\Phi(A_1, A_2)\},$$

$$\Sigma^H_{Ad} = \{\epsilon\},$$

$$S_{Ad} = S_{A_1} \times S_{A_2} \times (\Phi(A_1, A_2) \cup \{null\}) \times 2^{\Sigma^I_{Ad}} \times 2^{\Sigma^O_{Ad}},$$

$$S^\text{int}_{Ad} = S^\text{int}_{A_1} \times S^\text{int}_{A_2} \times \{null\} \times \emptyset \times \emptyset,$$
Description of the adapter construction

The set $\tau_{Ad}$ are defined as follows:

1. If $s_1 \xrightarrow{\delta a} s_1' \in \tau_{A_1} \land a \in \Sigma_{A_1} \setminus \Sigma_{syn(A_1, A_2)}$
   then $(s_1, s_2, R, O, l) \xrightarrow{c} (s_1', s_2, R, O, l) \in \tau_{Ad}$.

2. If $s_1 \xrightarrow{\delta a} s_1' \in \tau_{A_1} \land s_2 \xrightarrow{\delta a} s_2' \in \tau_{A_2} \land R = null \land a \in \Sigma_{shr(A_1, A_2)}$
   then $(s_1, s_2, R, O, l) \xrightarrow{c} (s_1', s_2', R, O, l) \in \tau_{Ad}$.

3. If $s_1 \xrightarrow{l a} s_1' \in \tau_{A_1} \land \exists \alpha \in \Phi(A_1, A_2) \text{ s.t. } a \in \Pi_1(\alpha) \land a \not\in O \land (R = null \lor R = \alpha)$
   then
   - $R' \leftarrow \alpha$, $O' \leftarrow O \cup \{a\}$,
   - If $O' = \Pi_1(\alpha)$ then $l' = \Pi_2(\alpha)$ else $l' = l$.
   - $(s_1, s_2, R, O, l) \xrightarrow{?a} (s_1', s_2, R', O', l') \in \tau_{Ad}$

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

5. Adapt points 1, 3 and 4 for the automaton $A_2$. 
Description of the adapter construction
Description of the adapter construction

Figure: Client ⊗ Adapter ⊗ Server
Adapter properties

By construction, the following properties hold:

1. *(No Mixing)*
   \[ \forall a \in \Sigma_{\text{shr}(A_1, A_2)}, \forall s = \langle s_1, s_2, R, O, I \rangle \text{ a state of Ad}, \]
   \[ \exists s_1 \xrightarrow{\delta_a} s'_1 \in \tau_{A_1} \text{ and } \exists s_2 \xrightarrow{\delta_a} s'_2 \in \tau_{A_2} \Rightarrow R = \text{null}, O = \emptyset \text{ and } I = \emptyset. \]

2. *(Coherence)*
   \[ \forall \sigma \text{ a sequence of Ad such that each of its state has its third field non null, except the states of the extremities, the following points hold:} \]
   - The intermediate states of \( \sigma \) are attached to a same rule, say \( \alpha \).
   - Each action of \( \alpha \) appears once and only once in \( \sigma \); the output actions of \( \alpha \) (i.e. which are necessarily input actions of Ad) appear before its input ones.
   - \( \epsilon \) actions may appear in \( \sigma \). 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.
Conclusion

- we presented simple and clear automata construction for software adaptation based on mapping rules.
- we opted for a *strong* synchronisation semantic, synchronisations between same components can not be interfered.

Future works:
- Developing tools to assist designers to generate adapters.
- extend component interfaces to take into account adaptation of components with temporal constraints and verification of temporal properties.
Merci.
Grigoris Antoniou, Grigoris Antoniou, Grigoris Antoniou, Frank Van Harmelen, and Frank Van Harmelen.
Web ontology language : Owl.

Antonio Brogi and Razvan Popescu.
Automated generation of bpel adapters.

S. Chouali, S. Mouelhi, and H. Mountassir.
Adapting components behaviours using interface automata.

Hamid Reza Motahari Nezhad, Boualem Benatallah, Axel Martens, Francisco Curbera, and Fabio Casati.
Semi-automated adaptation of service interactions.