A Petri Net Approach for Reusing and Adapting Components with Atomic and non-atomic Synchronisation

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

[1] MOVEP, USTHB, Algiers.
dzaouche,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 based on Petri nets which relies on mapping rules to generate automatically adaptors and check compatibilities of components. Our solution addresses both signature and behaviour level and covers both asynchronous and synchronous communication between components. State space of the Petri model is used to localise mismatches.

Keywords: Interface automata, components reuse, components adaptation Petri nets, synchronous and asynchronous communication.

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 [8, 9].
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, 3]. 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, (ii) at behaviour or protocol level as incompatible orderings of messages, and (iii) at semantic aspect concerning senses of operations as the use of synonyms for messages or methods [3].

There exist some works aiming at working out mismatches of components which remain incompatible, even in the optimistic approach. These works generally use adaptors, which are components that can be plugged between the mismatched components to convert the exchanged information causing mismatches. For example, the approach proposed in [11] operates at the implementation level by introducing data conversion services. Similarly, in [2, 7] smart data conversion tools are deployed to resolve data format compatibility issues during workflow composition.
Other works are based on formal methods such as interface automata, logic formula and Petri nets which give formal description to software interface and behaviour [3, 5].
In [4], 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 allows not atomic action synchronization, but doesn't cover all possible behaviours. In [3], manual adaptation contracts are used cutting off some incorrect behaviours. They propose two approaches based on interface automata and Petri nets, respectively. However, unlike our approach, these works allow only asynchronous communications. In [6] the behaviour of interacting components is modelled by labelled Petri nets where labels represent requested and provided services. The component models are composed in such a way that incompatibilities are manifested as deadlocks in the composed model. In [13], OR-transition Colored Petri Net is used to formalize and model components where transitions can effectively represent the operations of the software component. Both [6] and [13] focus more on component composition than on adaptation.
In our approach, we propose Petri net construction to check compatibilities of components according to a set of matching rules without any behaviour restriction. Contrary to [3], we deal with both synchronous and asynchronous communications. We use state graph of the Petri net model to localise mismatches.
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 by the component, that means methods that can be called or reception of messages. 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.

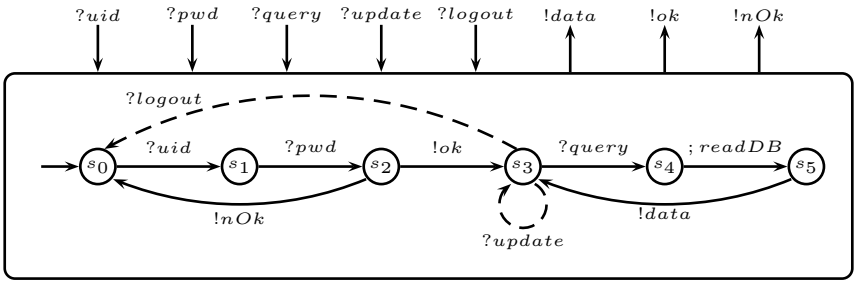
Definition 1 *(Interface automaton)*
An interface automaton $A = \langle S_A, S_A^{init}, \Sigma_A, \tau_A \rangle$ *where :*

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

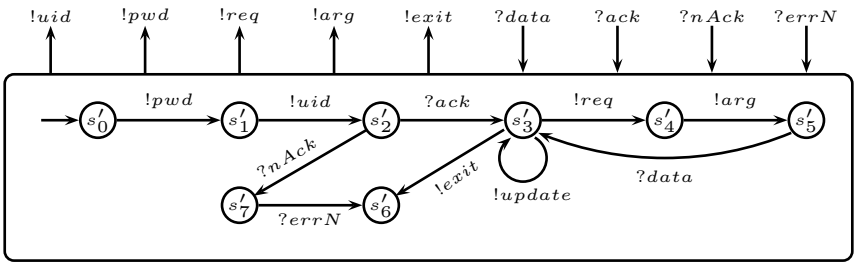
The input or output actions of automaton A are called external actions denoted by $\Sigma_A^{ext} = \Sigma_A^O \cup \Sigma_A^I$. A is closed if it has only internal actions, that is $\Sigma_A^{ext} = \emptyset$; otherwise we say that A is open. Input, output and internal actions are respectively labelled by the symbols "?", "!" and ";". 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 model 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 structured into a password* (!*pwd*) *and a username* (!*uid*)*. If Client is not authenticated by Server* (!*nAck and* !*errN*)*, it exits. Otherwise, Client loops on sending read or update requests. A read request* (!*req*) *is followed by its parameters* (!*arg*)*, then Client waits the result* (?*data*)*. An update request is an atomic action* (!*update*)*. At any moment, Client can exit* (!*exit*)*.*
On the other hand, when Server receives a username (?*uid*) *followed by a password* (?*pwd*)*, 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* (;*readDB*) *and sends the appropriate data* (!*data*)*. Server can execute an update request* (?*update*)*.*
Figure 1.a depicts interface automaton Server. It is composed of six states $(s_0, \ldots s_5)$*, with state* s_0 *being initial, and nine steps, for instance* $(s_0, ?uid, s_1)$.

Some arcs are dashed, they will be referred in section 4. The sets of input, output and internal actions are given below:
- Σ^O_{Server} =$\{ok, nOk, data\}$,
- Σ^I_{Server} = $\{uid, pwd, logout, query, update\}$,
- Σ^H_{Server}= $\{readDB\}$.



(a) Server



(b) Client

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

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 $shared(A_1, A_2) = (\Sigma^I_{A_1} \cap \Sigma^O_{A_2}) \cup (\Sigma^O_{A_1} \cap \Sigma^I_{A_2})$, e.g. set $Shared(Client, Server) = \{uid, pwd, update, data\}$.
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. $\Sigma^H_{A_1 \otimes A_2} = (\Sigma^H_{A_2} \cup \Sigma^H_{A_1}) \cup shared(A_1, A_2)$,
4. $\Sigma^I_{A_1 \otimes A_2} = (\Sigma^I_{A_1} \cup \Sigma^I_{A_2}) \setminus shared(A_1, A_2)$,
5. $\Sigma^O_{A_1 \otimes A_2} = (\Sigma^O_{A_1} \cup \Sigma^O_{A_2}) \setminus shared(A_1, A_2)$,
6. $\tau_{A_1 \otimes A_2} = \{(v,u), a, (v',u) \mid (v,a,v') \in \tau_{A_1} \wedge a \notin shared(A_1,A_2) \wedge u \in S_{A_2}\}$
   $\cup \{(v,u), a, (v,u') \mid (u,a,u') \in \tau_{A_2} \wedge a \notin shared(A_1,A_2) \wedge v \in S_{A_1}\}$
   $\cup \{(v,u), a, (v',u') \mid (v,a,v') \in \tau_{A_1} \wedge (u,a,u') \in \tau_{A_2} \wedge a \in shared(A_1,A_2)\}$.

An action of $Shared(A_1, A_2)$ 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 not shared input (resp. output) actions of A_1 or A_2 are input (resp. output) ones for $A_1 \otimes A_2$ (4, 5). Each state of the product consists of a state of A_1 together with a state of A_2 (1). 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 [1].
The automata associated with *Client* and *Server* are composable since definition 2 holds. However, their synchronous product is empty, in fact (s_0, s'_0) is an illegal state: *Client* sends password ($!pwd$) while *Server* requires a username ($?uid$), causing a deadlock situation. Thus, *Client* and *Server* are incompatible. As mentioned in the introduction, two incompatible components can be composed provided that there exits an adaptor 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 [11].

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 $(L_1 \cup L_2) \cap shared(A_1, A_2) = \emptyset$ 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 Client a read request is structured into two parts (!req and !arg), whereas it is viewed as one part (?query) in Server. 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:*

- $\Phi_{(Client,Server)} = \{\alpha_1, \alpha_2, \alpha_3, \alpha_4\}$ *with :*
  $\alpha_1 = (\{!req, !arg\}, \{?query\})$,
  $\alpha_2 = (\{?ack\}, \{!ok\})$,
  $\alpha_3 = (\{?nAck, ?errN\}, \{!nOk\})$,
  $\alpha_4 = (\{!exit\}, \{?logout\})\}$
- $\Sigma_{\Phi(Client,Server)} =$
  $\{req, arg, query, ack, ok, nAck, nOk, errN, exit, logout\}$

4 Towards Components 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. In fact, $A_1 \otimes A_2$ doesn't deal with correspondence between actions of $\Sigma_{\Phi(A_1,A_2)}$. Moreover, the product $A_1 \otimes A_2$ doesn't accept shared actions which have incompatible ordering in A_1 and A_2. For instance *Client* sends a password followed by a user name, whereas *Server* accepts the last message and then the former one. It is obvious that $A_1 \otimes A_2$ cannot be used to check the compatibility of A_1 and A_2. In this context, an adaptor component, must be defined. Such an adaptor is mainly based on the set $\Phi(A_1, A_2)$ and is a mediator between A_1 and A_2. It receives the output actions specified in $\Sigma_{\Phi(A_1,A_2)}$ from one automaton and sends the corresponding input actions to the other. In case of incompatible ordering of shared actions, the adaptor works out such situations by receiving, reordering and sending such actions to their destination component.

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.

4.1 Petri Net Construction for Components Adaptation

Contrary to interface automata formalism, the Petri net model is well suited to validate interactions between components, especially whenever events reordering is required. In fact, Petri nets allow to store resources (e.g. messages) before their use. In this paper, we use a Petri net model to adapt two interface automata according to a set of mapping rules given by the user of the components. The approach we propose consists of building a Petri net which mimics the component interfaces. Furthermore, the Petri net also contains a set of transitions, one per matching rule, which represent the adaptor component. More details will be given below.

First, we give the basic definitions of a Petri net model. For more details, we refer reader to [12, 10].

Definition 6 *(Labeled Petri Net) A Petri net N is a tuple $\langle P, T, W, \lambda \rangle$ where :*

- *P is a set of places,*
- *T is a set of transitions such that $P \cap T = \emptyset$,*
- *W is the arc weight function defined from $P \times T \cup T \times P$ to $\mathbb{N}$.*
- *λ is a label mapping defined from T to an alphabet set $\Sigma \cup \{\epsilon\}$.*

A marking is a function M: $P \rightarrow \mathbb{N}$ where $M(p)$ denotes the number of tokens at place p. The firing of a transition depends on enabling conditions.

Definition 7 *(Enabling) A transition t is enabled in a marking M iff $\forall\ p \in P$, $M(p) \geq W(p, t)$.*

Definition 8 *(Firing rule in a Marking) Let t be a transition enabled in a marking M. Firing t yields a new marking M', $\forall p \in P$, $M'(p) = M(p) - W(p, t) + W(t, p)$.*

Definition 9 *(State Space) A state space, denoted by $\mathbb{S}(N, M_0)$, of a marked labelled Petri net (N, M_0) is an oriented graph of accessible markings starting from M_0. An arc $M \xrightarrow{t} M'$ of $\mathbb{S}(N, M_0)$ means that M' is obtained by firing t from M.*

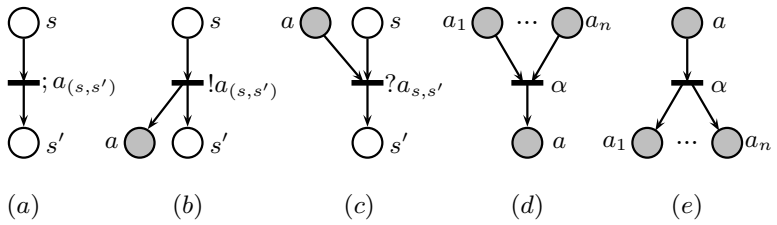


Fig. 2: Translation rules

The algorithm described below returns a marked labelled Petri net (N, M_0) composed of three parts dedicated for A_1, A_2 and a set of matching rules. These parts are glued by mean of places which model communication channels and are associated with external actions of A_1 and A_2, i.e. the actions of sets $Shared(A_1, A_2)$ and $\Sigma_{\Phi(A_1,A_2)}$.

For each state s (resp. external action a) of A_1 and A_2, the algorithm generates a corresponding place s (resp. a) in N. Furthermore, the places corresponding to initial states of interface automata will be initially marked in N.

Fig. 2.a, 2.b and 2.c show how to translate steps of A_1 and A_2. The gray full circles represent communication places. An internal action $s \xrightarrow{;a} s'$ is represented by a transition $;a_{(s,s')}$ which has an input place s and an output place s' (Fig 2.a). An output action $s \xrightarrow{!a} s'$ is represented by a transition $!a_{(s,s')}$ which has an input place s and two output places s' and a. A firing of $!a_{(s,s')}$ produces a token in place a modelling an emission of a message a (see Fig 2.b). Fig 2.c gives the translation of an input action $s \xrightarrow{?a} s'$, here each firing of $?a_{(s,s')}$ models a reception of a message a.

Fig. 2.d and 2.b show how to translate the mismatch rules. For each mismatch rule $\alpha = (\{!a\}, \{?a_1, \ldots, ?a_n\})$ of $\Phi(A_1, A_2)$, a transition α is added. The input places of α are $a_1 \ldots a_n$ and its output place is a. Each firing of α models the receptions of $a_1 \ldots a_n$ and the emission of a (see Fig 2.d). The same pattern is applied for a rule $\alpha = (\{?a_1, \ldots, ?a_n\}, \{!a\})$. Fig 2.e shows the translation of a rule $\alpha = (\{!a_1, \ldots, !a_n\}, \{?a\})$ or $\alpha = (\{?a\}, \{!a_1, \ldots, !a_n\})$. In this case, each firing of α models the reception of a and the emissions of $a_1 \ldots a_n$. These transitions simulate the adaptor.

For analysis requirement (next section), transitions associated with mismatch rules are labelled by the rule names and the others by the corresponding action names.

Algorithm 1 *BuildPetriNet*

`Inputs` $A_1 = \langle S_1, A_1, I_1, T_1 \rangle$, $A_2 = \langle S_2, A_2, I_2, T_2 \rangle$ `and` $\Phi(A_1, A_2)$ `a set of rules`
`Output`

A labelled Petri Net $N = \langle P, T, W, \lambda \rangle$ and its initial marking M_0
Initialization $P = \emptyset$, $T = \emptyset$
Begin
// Generation of places corresponding to the states of A_1 *and* A_2
for each state $s \in S_1 \cup S_2$ do
  add a place s to P
  If $s \in S_{A_1}^{init} \cup S_{A_2}^{init}$ then $M_0(s) = 1$ else $M_0(s) = 0$
  Endif
endfor
// Places simulating direct communication between A_1 *and* A_2
for each action $a \in Shared(A_1, A_2)$ do
  add a place a to P
endfor
// Places simulating indirect communication between A_1 *and* A_2
for each action $a \in \Sigma_{\Phi(A_1,A_2)}$ do
  add a place a to P
endfor
// Transitions simulating steps of A_1 *and* A_2
for each transition $s_1 \xrightarrow{\delta a} s_2 \in T_1 \cup T_2$, (with $\delta \in \{!, ?, ;\}$)
  add a transition $\delta a_{s_1,s_2}$ to T
  $\lambda(\delta a_{s_1,s_2}) = \delta a$
  add the arcs $s_1 \rightarrow \delta a_{s_1,s_2}$ and $\delta a_{s_1,s_2} \rightarrow s_2$ to W
  case:
    $\delta = '!'$: add the arc $!a_{s_1,s_2} \rightarrow a$ to W
    $\delta = '?'$: add the arc $a \rightarrow ?a_{s_1,s_2}$ to W
  endcase
endfor
// Transitions simulating adaptor of A_1 *and* A_2
for each $\alpha \in \Phi(A_1, A_2)$ do
  add a transition α to T
  $\lambda(\alpha) = \alpha$
  case:
    $\alpha \in \{(\{!a\}, \{?a_1, \ldots, ?a_n\}), (\{\{?a_1, \ldots, ?a_n\}, \{!a\})\}$:
      add the arcs $a_i \rightarrow \alpha$, $i \in 1 \ldots n$, and $\alpha \rightarrow a$ to W,
    $\alpha \in \{(\{!a_1, \ldots, !a_n\}, \{?a\}), (\{?a\}, \{!a_1, \ldots, !a_n\})\}$:
      add the arcs $a \rightarrow \alpha$ and $\alpha \rightarrow a_i$, $i \in 1 \ldots n$, to W
  endcase
endfor
return (N, M_0)
End

Fig. 3 gives a labelled and marked Petri net N associated with *Client* and *Server* according to the set of rules $\Phi(Client, Server)$ (which are defined in example 2). For sake of clarity, communication places are duplicated and transitions are represented by their labels. Moreover, transitions ?*update*, !*update*

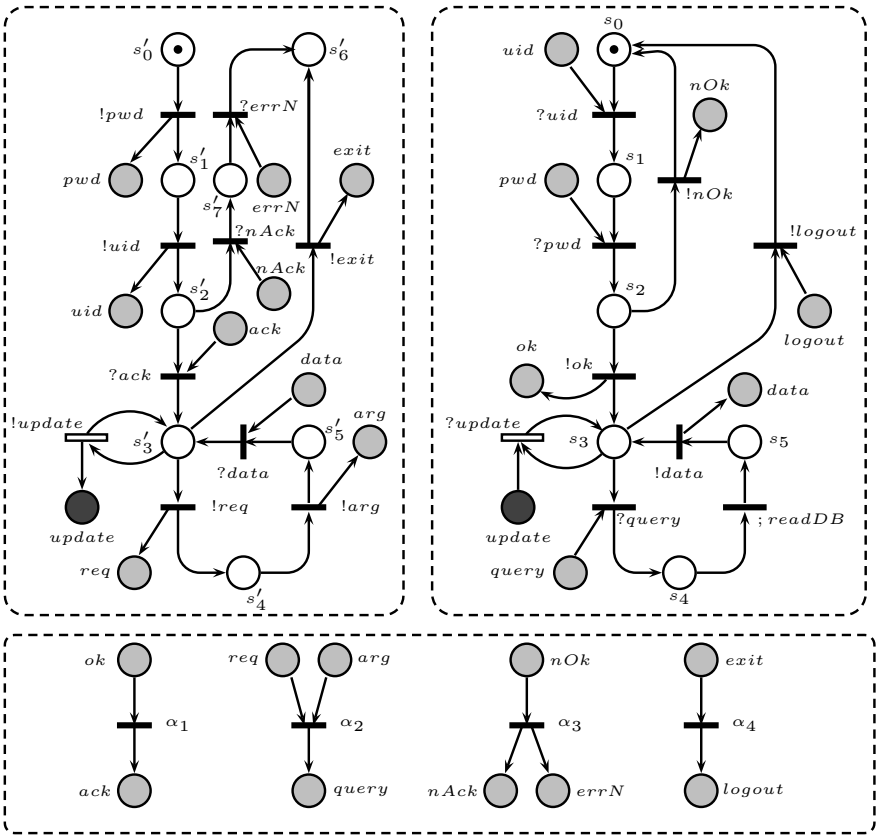


Fig. 3: A Petri net for adaptation of *Server* and *Client*

and place *update* are represented differently, a special attention will be accorded to them in the next section. The left and right parts of the net are respectively dedicated to *Client* and *Server*, they are glued by mean of communication places *uid*, *pwd*, *update* and *data*. These latter correspond to the actions of $Shared(Client, Server)$ and are used to simulate direct communications between *Client* and *Server*.

The lower part of the net represents the *adaptor*, it contains four transitions $\alpha_1, \alpha_2, \alpha_3$ and α_4, each one represents a rule of $\Phi(Client, Server)$. The communication places *req*, *arg*, *query*, *ack*, *ok*, *nAck*, *nOk*, *errN*, *exit* and *logout* are used to link the three parts and correspond to the actions of $\Sigma_{\Phi(Client,Server)}$.

Places s_0 and s'_0 are initially marked in N, they translate the initial places of *Client* and *Server* automata.

4.2 Synchronisation Semantics between Components

At this level, the Petri net construction models only asynchronous communication between two components. Such kind of communication may be source of incoherence as illustrated by the following scenario:

- *Client*: Authentication,
- *Server*: Okay message,
- *Client*: update request,
- *Client*: read request,
- *Server*: response for the read request,
- *Server*: data base update.

It is worth noting that the result of the read request may be incorrect. This occurs whenever the required information is concerned by the *update* operation. To work out this problem, *Client* and *Server* must synchronise on *update* action. Therefore, we propose to enrich the Petri net construction to strengthen synchronisation between transitions which are related to critical shared actions (e.g. *update* action): (1) such transitions *must be fired by pair* (w.r.t some critical action, one for an output step and the other for an input step). (2) The communication places of critical external actions are not useful since here messages are not stored. (3) The set of critical actions, denoted by $Synch$, is an input of the algorithm. The set of transitions related to $Synch$ is denoted by T_{Synch}. For instance, to avoid the previous scenario, action *update* is considered as critical, so transitions $!update$ and $?update$ must be fired simultaneously. Place *update* is omitted, $Sync = \{update\}$ and $T_{Sync} = \{?update, !update\}$.

4.3 Building and analysing state space

In order to model synchronous communication between components, transitions of T_{Synch} are fired by pair. Further conditions are necessary to fire simultaneously a pair of transitions t and t' belonging to T_{Synch} from a state s :

- $\lambda(t) = \delta a$ and $\lambda(t') = \overline{\delta} a$.
- Both t and t' are enabled in s.

As mentioned in the introduction, the compatibility control of components is made by using the state graph. In order to do this, we adapt the notion of illegal state for our approach. We use the classical definition [1] where an illegal state indicates that some service is required by one component but cannot be offered by the other one.

Definition 10 *(Illegal state)*
Let s be a state of $\mathbb{S}(N, M_0)$, s is an illegal state if:
- s has no successor and contains at least a marked communication place,
- or there is an enabled transition t of T_{synch}, with $\lambda(t) = !a$ but no enabled transition t' with $\lambda(t') = ?a$ in s.

The state graph of the marked Petri net shown in Fig. 3 contains no illegal state, therefore *Client* and *Server* can be composed according to the set of rules $\Phi(Client, Server)$.

Example 3

Consider again the example of Fig. 2 and let us omit the dashed arcs. The corresponding state graph contains two illegal states. Fig. 4 exhibits a particular sequence of the state graph, containing the two illegal states (gray states):

1. In $(s_3 s'_3)$, transition $!update$ is enabled but cannot be fired since transition $?update$ is not enabled within the state. This means that *Client* issues an update request which is not assumed by *Server* at this state.
2. State $(s_3 s'_6, exit)$ has no successor in the state graph and a marked communication place ($exit$). Such a mark means that *Client* has sent an *exit* request which will not be covered by *Server*.

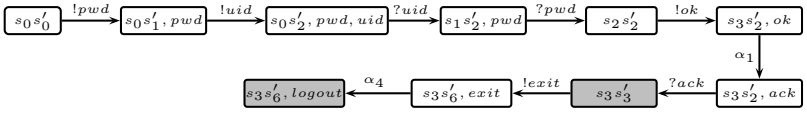


Fig. 4: A firing sequence

5 Conclusion

Software Adaptation is widely used for adapting incompatible components, viewed as black boxes. In this paper, we have presented a Petri net construction for software adaptation at signature and behavioural levels based on mapping rules. These latter are used to express correspondence between actions of components. The Petri net construction reflects the structure of component interface automata to assemble and their corresponding mapping rules. The proposed construction is incremental, e.g. rules can be easily added or replaced. Our approach allows both synchronous and asynchronous communications, unlike the other approaches referred in this paper. In our future work, we plane to extend our Petri net construction to take into account adaptation of components with temporal constraints.

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. S. Bowers and B. Ludascher. An ontology-driven framework for data transformation in scientific workflows. *DATA INTEGRATION IN THE LIFE SCIENCES, PROCEEDINGS*, 2994:1–16, 2004.
3. C. Canal, P. Poizat, and G. Salaun. Model-based adaptation of behavioral mismatching components. *IEEE Transactions on Software Engineering*, 34(4):546–563, 2008.
4. 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.
5. 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.
6. D. C. Craig and W. M. Zuberek. Petri nets in modeling component behavior and verifying component compatibility. In *Int. Workshop on Petri Nets and Software Engineering, in conjunction with the 28-th Int. Conf. on Applications and Theory of Petri Nets and Other Models of Concurrency*, 2007.
7. W. Kongdenfha, H.R. Motahari Nezhad, B. Benatallah, F. Casati, and R. Saint-Paul. Mismatch patterns and adaptation aspects: A foundation for rapid development of web service adapters. *IEEE Transactions on Services Computing*, 2(2):94–107, 2009.
8. C.W. Krueger. Software reuse. *ACM Comput. Surv.*, 24(2):131–183, June 1992.
9. L. Kung-Kiu and W. Zheng. Software component models. *IEEE Transactions on Software Engineering*, 33(10):709–724, 2007.
10. T. Murata. Petri Nets: Properties, Analysis and Applications. *Proceedings of the IEEE*, 77(4):541–580, 1989.
11. 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.
12. W. Reisig. *Understanding Petri Nets: Modeling Techniques, Analysis Methods, Case Studies*. Springer, 31 July 2013. 230 pages; ISBN 978-3-642-33277-7.
13. Yong Yu, Tong Li, Qing Liu, and Fei Dai. Modeling software component based on extended colored petri net. In Ran Chen, editor, *Intelligent Computing and Information Science*, volume 135 of *Communications in Computer and Information Science*, pages 429–434. Springer Berlin Heidelberg, 2011.