Synchronizing Behavioural Mismatch in Software Composition*

Carlos Canal[1], Pascal Poizat[2], and Gwen Salaün[3]

[1] University of Málaga, Department of Computer Science
Campus de Teatinos, 29071 Málaga, Spain
canal@lcc.uma.es
[2] IBISC FRE 2873 CNRS – University of Évry Val d'Essonne, Genopole
Tour Évry 2, 523 place des terrasses de l'Agora, 91000 Évry, France
Pascal.Poizat@ibisc.univ-evry.fr
[3] VASY project, INRIA Rhône-Alpes, France
655 avenue de l'Europe, 38330 Montbonnot Saint-Martin, France
Gwen.Salaun@inrialpes.fr

Abstract. Software Adaptation is a crucial issue for the development of a real market of components promoting software reuse. Recent work in this field has addressed several problems related to interface and behavioural mismatch. In this paper, we present our proposal for software adaptation, which builds on previous work overcoming some of its limitations, and makes a significant advance to solve pending issues. Our approach is based on the use of synchronous vectors and regular expressions for governing adaptation rules, and is supported by dedicated algorithms and tools.

1 Introduction

Component-Based Software Engineering (CBSE) focuses on composition and reuse, aiming to develop a market of software components, in which customers select the most appropriate software piece depending on its technical specification [6]. The development of such a market has always been one of the major concerns of Software Engineering, but it has never become a reality. The reason is that we cannot expect that any given software component perfectly matches the needs of a system where it is trying to be integrated. Software is never reused "as it is", especially in case of legacy code, and a certain degree of adaptation is always required [16].

To deal with these problems a new discipline, *Software Adaptation*, which is emerging, is concerned with providing techniques to arrange already developed pieces of software, in order to reuse them in new systems [7]. Software Adaptation promotes the use of *adaptors* —specific computational entities guaranteeing that components will interact in the right way.

* This work has been partly funded by the European Network of Excellence on AOSD, AOSD-Europe IST-2-004349-NOE.

R. Gorrieri and H. Wehrheim (Eds.): FMOODS 2006, LNCS 4037, pp. 63–77, 2006.

CBSE postulates that a component must be reusable from its interface [20], which in fact constitutes its full technical specification. Hence, we have to provide components with a specification that helps in the process of adapting and reusing them. The intended adaptation will then take the form of a mapping among the interface descriptions of the components involved.

The characteristics and expressiveness of the language used for interface description determines the degree of interoperability we can achieve using it, and the kind of problems that can be solved. We can distinguish between several levels of interoperability, and accordingly of interface description [8]: signature level (service names and types), behavioural level (interaction protocols), semantic level (functional specification of what the component actually *does*) and service level (non functional properties such as quality of service). At each one, mistmatch can occur [8] and would have to be corrected. Currently, industrial component models only tackle the signature level, with Interface Description Languages (IDLs). Although (automatic) adaptation in the semantic and service levels still remains uncertain, several approaches have been presented for extending component interfaces with behaviour, thus resulting in what we may call a Behavioural IDL (BIDL) (*e.g.*, WSBPEL [1] for web services).

In this paper, we focus on mismatch appearing at the behavioural level. Intuitively, it means that two (or more) components cannot —as they are— interact till they reach correct termination states. To compensate such behavioural incompatibilities, we propose first to use synchronous vectors as the mapping language to make explicit communications on different message names. Second, we extend our notation to enable writing regular expressions of vectors. Such a mapping notation is convenient to describe in an abstract way more advanced adaptation scenarios such as reordering of messages. Figure 1 gives a graphical overview of our method for adaptation.

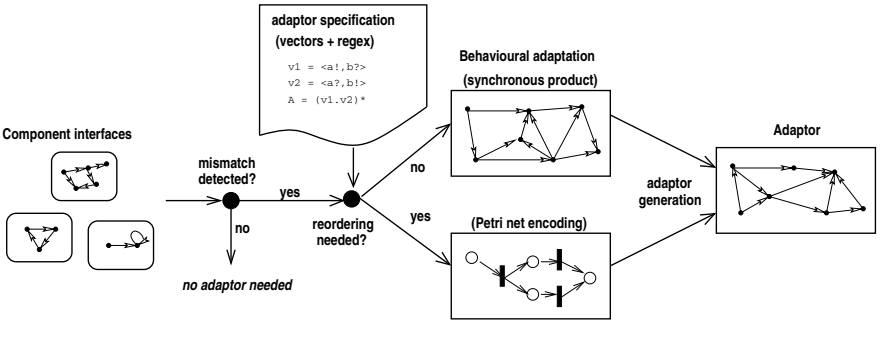

Fig. 1. Overview of our approach for adaptation of incompatible components

The remainder of the paper is organized as follows. Section 2 formally introduces our component interface model, and defines interface mismatch by means of synchronous products. Section 3 presents our approach to component adaptation, which combines the points in favour of different adaptation approaches,

while trying to overcome their limitations. Our proposals for behavioural adaptation with or without message reordering are supported by dedicated algorithms, and in both cases the adaptation mappings rely on synchronous vectors. Next, Section 4 extends our initial mapping notation with regular expressions, enabling complex policies for applying the adaptation vectors. In Section 5, we survey the more advanced proposals for software adaptation, and compare ours to them. Finally, Section 6 draws up the main conclusions of this work and sketches some future tasks that will be accomplished to extend its results.

2 Interfaces and Mismatch

2.1 Component Interfaces

Component interfaces are given using a signature and a behavioural interface.

Definition 1 (Signature). *A* signature Σ *is a set of operation profiles. This set is a disjoint union of* provided *operations and* required *operations. An operation profile is simply the name of an operation, together with its argument types, its return type and the exceptions it raises.*

This definition naturally corresponds to the signature definitions in component based models such as CCM or J2EE. Such signatures are defined using an IDL. For the sake of simplicity in the presentation, in this paper we do not deal with operation arguments, return values or exceptions.

We also take into account behavioural interfaces through the use of Labelled Transition Systems (LTSs).

Definition 2 (LTS). *A* Labelled Transition System *is a tuple* (A, S, I, F, T) *where:* A *is an alphabet (set of events),* S *is a set of states,* $I \in S$ *is the initial state,* $F \subseteq S$ *are final states, and* $T \subseteq S \times A \times S$ *is the transition function.*

The alphabet of the LTS is built on the signature. This means that for each provided operation p in the signature, there is an element $p?$ in the alphabet, and for each required operation r, an element $r!$. As in CCS, $(a, \bar{a})$ denote complementary actions —*i.e.*, if a is $p?$ (respectively $r!$), then $\bar{a}$ is $p!$ (respectively $r?$).

LTSs are adequate as far as user-friendliness and development of formal algorithms are concerned. However, higher-level behavioural languages such as process algebras can be used to define behavioural interfaces in a more concise way. In this paper, we use as a BIDL the part of the CCS notation restricted to sequential processes which can be translated into LTS models: `P ::= 0 | a?.P | a!.P | P1+P2 | A`, where `0` denotes a do-nothing process, `a?.P` a process which receives `a` and then behaves as `P`, `a!.P` a process which sends `a` and then behaves as `P`, `P1+P2` a process which may act either as `P1` or `P2`, and `A` denotes the call to a process defined by an agent definition equation `A = P`.

As process algebras do not enable to define initial and final states, we extend this CCS notation to tag processes with initial (`i`) and final (`f`) attributes. Finally, `0` is often omitted in processes (*e.g.*, `a!.b![f]` is used for `a!.b!.0[f]`).

Example 1. Consider a client that repetitively sends a query and its argument, and then waits for an acknowledgement, quitting with an `end!`, and a server repetitively waiting for a query and a value, then returning a given service:

```
Client[i] = query!.arg!.ack?.Client + end![f]
Server[i,f] = query?.value?.service!.Server
```

The LTSs for these two components are given below with initial and final states respectively marked by input arrows and black circles.

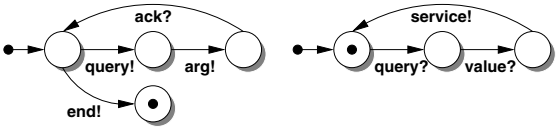

Fig. 2. A simple client/server system

2.2 Behavioural Mismatch

Various definition of behavioural mismatch have been proposed in the field of software adaptation and software architecture analysis [8]. We build on the most commonly accepted one, namely deadlock-freedom. The first step is to define the semantics of a system made up of several identified components. This semantics can be given, following work by Arnold [2] using synchronous product.

Definition 3 (Synchronous Product). *The* synchronous product *of n LTSs* $L_i = (A_i, S_i, I_i, F_i, T_i)$*,* $i \in 1..n$*, is the LTS* (A, S, I, F, T) *such that:*

- $A \subseteq \Pi_{i \in 1..n} A_i$*,* $S \subseteq \Pi_{i \in 1..n} S_i$*,* $I = (I_1, \ldots, I_n)$*,*
- $F \subseteq \{(s_1, \ldots, s_n) \in S \mid \bigwedge_{i \in 1..n} s_i \in F_i\}$*,*
- T *is defined using the following rule:*
  $\forall (s_1, \ldots, s_n) \in S$*,* $\forall i, j \in 1..n, i < j$ *such that*
  $\exists (s_i, a, s'_i) \in T_i$*,* $\exists (s_j, \bar{a}, s'_j) \in T_j$*, then*
  $(x_1, \ldots, x_n) \in S$ *and* $((s_1, \ldots, s_n), (l_1, \ldots, l_n), (x_1, \ldots, x_n)) \in T$*, where*
  $\forall k \in 1..n$*,* $l_k = \{\ a$ if $k = i$*,* $\bar{a}$ if $k = j$*,* ε *otherwise* $\}$
  $x_k = \{\ s'_i$ if $k = i$*,* s'_j if $k = j$*,* s_k *otherwise* $\}$

We are now able to characterize behavioural mismatch by means of deadlock.

Definition 4 (Deadlock State). *Let* $L = (A, S, I, F, T)$ *be an LTS. A state* s *is a deadlock state for* L*, noted* $\mathtt{dead}(s)$*, iff it is in* S*, not in* F *and has no outgoing transitions:* $s \in S \wedge s \notin F \wedge \nexists l \in A, s' \in S\ .\ (s, l, s') \in T$*.*

Definition 5 (Deadlock Mismatch). *An LTS* $L = (A, S, I, F, T)$ *presents a deadlock mismatch if there is a state* s *in* S *such that* $\mathtt{dead}(s)$*.*

To check if a system made up of several components presents behavioural mismatch, its synchronous product is computed and then Definition 5 is used.

Fig. 3. Synchronous product for the client/server system in Figure 2

Example 2. Taking Example 1, we obtain the following synchronous product:

Note that the deadlock is caused by (i) the client required service `end!` which has no counterpart in the server, and (ii) name mismatching between the client required service `arg!` and the server provided service `value?`.

We may now define what is a correct adaptor for a system. An adaptor is given by an LTS which, put into a non-deadlock-free system yields a deadlock-free one. For this to work, the adaptor has to preempt all the component communications. Therefore, prior to the adaptation process, component service names may have to be renamed prefixing them by the component name, *e.g.*, `c:service!`.

The product we have defined here is common in the community and hence is supported by tools such as the CADP toolbox [9]. Our deadlock definition however is slightly different from the one used in these tools, since it has to distinguish between success (deadlock in a final state), and failure (deadlock in a non-final state). Mismatch detection can be automatically checked by CADP up to the adding within component interfaces of specific loop transitions labelled with `accept` over final states. Then the EXP.OPEN tool [13] of CADP is used to perform a full matching product between the component interfaces.

3 Adaptation Based on Synchronous Vectors

3.1 Synchronizing with Vectors

The first thing to solve in adaptation is impossible communication due to different event/message names. Our idea is to use synchronous vectors as a way to denote a morphism between event names in different components.

Vectors generalize synchronous product by expressing not only synchronization between processes on the same event names (a and $\bar{a}$ in Definition 3), but more general correspondences between the events of the process involved.

Definition 6 (Vector). *A* synchronous vector *(or* vector *for short) for a set of* Id *indexed components* $L_i = (A_i, S_i, I_i, F_i, T_i)$, $i \in Id$, *is a tuple* (e_i) *with* $e_i \in A_i \cup \{\varepsilon\}$, ε *meaning that a component does not participate in a synchronization.*

Note that vectors are simple correspondences between events. Extensions can be easily defined to consider relations between events with data.

Definition 7 (Synchronous Vector Product). *The* synchronous vector product *of* n *LTSs* $L_i = (A_i, S_i, I_i, F_i, T_i)$, $i \in 1..n$ *with a set of vectors* V, *is the LTS* (A, S, I, F, T), *denoted by* $\Pi(L_i, V)$, *such that:*

- $A \subseteq \Pi_{i \in 1..n} A_i$, $S \subseteq \Pi_{i \in 1..n} S_i$, $I = (I_1, \ldots, I_n)$,
- $F \subseteq \{(s_1, \ldots, s_n) \in S \mid \bigwedge_{i \in 1..n} s_i \in F_i\}$,

– *T is defined using the following rule:*
$((s_1,\ldots,s_n),(l_1,\ldots,l_n),(s'_1,\ldots,s'_n)) \in T$ and $(s'_1,\ldots,s'_n) \in S$ if
$\exists (s_1,\ldots,s_n) \in S$ and $\exists v = (l_1,\ldots,l_n) \in V$ such that,
$\forall l_i \in v$ $s'_i = s_i$ if $l_i = \varepsilon$ and $\exists (s_i, l_i, s'_i) \in T_i$ otherwise.

3.2 Behavioural Adaptation Without Reordering

We first address adaptation where only event names mismatch is taken into account, that is impossible communications due to different message names. Our algorithm takes as input the Id indexed set of components LTSs L_i of the systems and a mapping which is a synchronous vector V.

1. compute the product $P = (A_P, S_P, I_P, F_P, T_P) = \Pi(L_i, V)$
2. obtain $P_{\text{restr}} = (A_{P_{\text{restr}}}, S_{P_{\text{restr}}}, I_{P_{\text{restr}}}, F_{P_{\text{restr}}}, T_{P_{\text{restr}}})$ from P recursively removing transitions and states yielding deadlocks: find a state s such that $\texttt{dead}(s)$, remove s and any transition t with target s, and do this until there is no more such s in the LTS.
3. from P_{restr}, build the adaptor $A = (A_{P_{\text{restr}}}, S_{P_{\text{restr}}} \cup S_{\text{add}}, I_{P_{\text{restr}}}, F_{P_{\text{restr}}}, T_A)$ where S_{add} and T_A are defined as follows.
For each $t = (s = (s_1,\ldots,s_n),(l_1,\ldots,l_n),s' = (s'_1,\ldots,s'_n))$ in $T_{P_{\text{restr}}}$, let $L_{\text{rec}} = \{l? \mid l! \in (l_1,\ldots,l_n)\}$ and $L_{\text{em}} = \{l! \mid l? \in (l_1,\ldots,l_n)\}$. Let then Seq_{rec} be the set of all permutations over L_{rec} and Seq_{em} be the set of all permutations over L_{em}. For each couple (R,E) in $\text{Seq}_{\text{rec}} \times \text{Seq}_{\text{em}}$, $R = (r_1,\ldots,r_{nr})$ and $E = (e_1,\ldots,e_{ne})$, $\text{seq} = (r_1,\ldots,r_{nr},e_1,\ldots,e_{ne})$, construct the transaction

$$s = q_0 \overset{\text{seq}[1]}{\rightarrow} q_1 \ldots q_k \overset{\text{seq}[k+1]}{\rightarrow} q_{k+1} \ldots q_{n-1} \overset{\text{seq}[n]}{\rightarrow} s' = q_n$$

adding each $q_{k\in 1..n-1}$ in S_{add} and each $q_k \overset{\text{seq}[k+1]}{\rightarrow} q_{k+1}$ $(k \in 0..n)$ in T_A.

This algorithm builds the most general adaptor in the sense that it simulates any other adaptor for the mismatching system. Its complexity lies mainly in the synchronous product construction $\mathbf{O}(|S|^n)$ where S is the largest set of states.

3.3 Behavioural Adaptation with Reordering

Let us now extend the domain of adaptation problems we deal with. The goal is to also address behavioural mismatch with reordering, that is, the incompatible ordering of the events exchanged. Indeed, our behavioural adaptation proposal above would yield an empty adaptor in presence of such behavioural mismatch, concluding that adaptation is not possible. In this case, the adaptation process may try to reorder protocol events in-between the components. To this purpose, we present a second approach which complements the first one. However, it does not replace it as the process may not agree on message reordering.

This behavioural adaptation approach is based on previous works dedicated to the analysis of component queue boundedness [14]. In order to accommodate

behavioural mismatch, the events received by the adaptor are de-synchronized from their emission. Our algorithm can be simulated by a translation of the problem into Petri nets [15]. The main advantage of such an approach is that it is equipped with efficient tools.

We first proceed by constructing a Petri net representation of the assumptions the components make on their environment (by mirroring their behavioural interfaces), and then build causal dependences between the events received and sent by the adaptor accordingly to the mapping, given under the form of synchronous vectors. This allows us to build an adaptor which accommodates both behavioural mismatch (with or without reordering).

1. for each component i with LTS L_i, for each state $s_j \in S_i$, add a place `Control-i-s_j`
2. for each component i with initial state I_i, put a token in `Control-i-I_i`
3. for each $a!$ in $\bigcup_i A_i$, add a place `Rec-a`
4. for each $a?$ in $\bigcup_i A_i$, add a place `Em-a`
5. for each component i with LTS L_i, for each $(s, l, s') \in T_i$:
   - add a transition with label l, one arc from place `Control-i-s` to the transition and one arc from the transition to place `Control-i-s'`
   - if `l` has the form $a!$ then add one arc from the transition to place `Rec-a`
   - if `l` has the form $a?$ then add one arc from place `Em-a` to the transition
6. for each vector $v = (l_1, \ldots, l_n)$ in V:
   - add a transition with label `tau`
   - for each l_i with form $a!$, add one arc from place `Rec-a` to the transition
   - for each l_i with form $a?$, add one arc from the transition to place `Em-a`
7. for each tuple $(f_1, \ldots, f_n)$, $f_i \in F_i$, of final states, add a (loop) `accept` transition with arcs from and to each of the tuple f_i

Once this Petri net encoding has been performed, we compute its marking graph. If it is finite (*e.g.*, for non recursive adaptors) then it gives a behavioural description of the adaptor. If not (it cannot be computed in finite time), then we compute the coverability graph of the net. Note that due to the overapproximation of such a graph, we add a guard `[#Em-a>1]` (`#Em-a` meaning the number of tokens in place `Em-a`) on any `a!` transition in this graph leaving a state where `#Em-a` is ω. In both cases (marking or coverability graph), step 2 of the algorithm in Section 3.2 has to be performed on the adaptor obtained. The complexity of this algorithm lies mainly in the marking or coverability graph construction which is exponential [17].

This algorithm is supported by tools. We have made successful experiments with the TINA tool [3] to generate marking and coverability graphs. Our approach yields graphs which can be too large for a human reader. We simplify the adaptor LTS passing the resulting output file to CADP and performing a $\tau * a$ reduction on it to remove the meaningless `tau` transitions it contains.

3.4 Application

We here present an example following the behavioural adaptation technique above.

Example 3. Suppose we have a client `Client[i]=req!.arg!.ack?[f]` and a server `Server[i]=value?.query?.service![f]` with vectors <`req!,query?`>, <`arg!,value?`> and <`ack?,service!`>. Such an example is typical of clients and servers which follow different standards for the order of sending subservice elements. The Petri net encoding (see Section 3.3) of the system is:

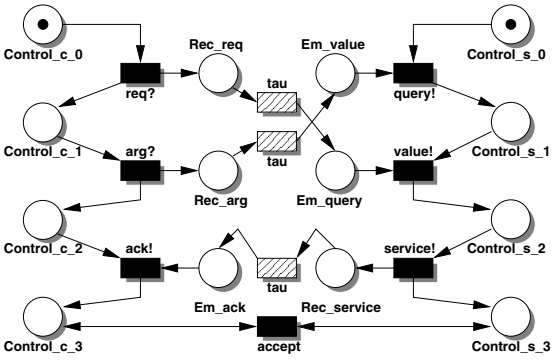

Fig. 4. Petri net encoding of a simple client/server system

Computing the marking graph, we obtain an LTS with 13 states and 16 transitions (Fig. 5, left), which once reduced yields the correct adaptor (Fig. 5, right)[1].

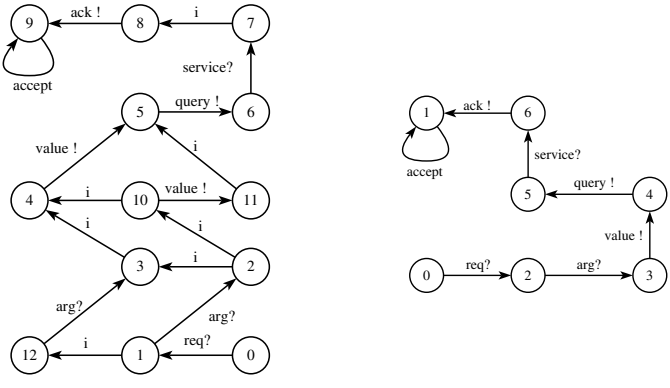

Fig. 5. Initial and reduced adaptor for the client/server system

We want to stress that our adaptation proposal is an *automatic* process. For the sake of the presentation, we have shown here a simple example for which the adaptor could be obtained manually. However, using slightly more complex

[1] Note the `i` which stands in CADP for `tau` transitions, and the `accept` loop transitions which enable the detection of correct final states.

component protocols, the adaptor becomes too large to be obtained by hand. Moreover, the use of regular expressions in the next section will increase the complexity of the adapting process and the need for such automatic techniques.

4 Adaptation Patterns

In this section, we tackle the problem of adaptation mappings which may change over time. In the following, we present a way to express such mappings using regular expressions (regex), and then update our algorithms to deal with them.

4.1 Regular Expressions (Regex) of Vectors

First, we introduce the syntax for regex. These will be used in place of the basic vector mappings we presented in Section 3.

Definition 8 (Vector Regex). *Given n LTSs $L_i = (A_i, S_i, I_i, F_i, T_i)$, and a set of vectors $V = \{(e_{ij})\}_j$ for their adaptation, with $e_{ij} \in A_i \cup \{\varepsilon\}$, a (vector) regex for these LTSs can be generated by the following syntax:* `R ::= v (VECTOR)| R1.R2 (SEQUENCE)| R1+R2 (CHOICE)| R* (ITERATION)`*, where* `R, R1, R2` *are regex, and* `v` *is a vector in V*

A graphical description such as LTS labelled with vectors might be used instead of regular expressions to favour readability and user-friendliness of the notation.

Example 4 (Alternating use client). Suppose we have a system formed by one client `C` and two servers, `S` and `A`:

```
C[i] = end![f] + req!.arg!.ack?.C,
S[i,f] = value?.query?.service!.S, and
A[i,f] = value?.query?.service!.A.
```

One may want to express in the adaptation mapping that the client accesses the two servers alternatively, and not always the same one. For this, we use the following regex: $(\mathtt{v}_{s1}.\mathtt{v}_{s2}.\mathtt{v}_{s3}.\mathtt{v}_{a1}.\mathtt{v}_{a2}.\mathtt{v}_{a3})*.\mathtt{v}_{\text{end}}$ with

$\mathtt{v}_{s1}$ =<req!,query?,ε>, $\mathtt{v}_{a1}$ =<req!,ε,query?>, $\mathtt{v}_{end}$ =<end!,ε,ε>,
$\mathtt{v}_{s2}$ =<arg!,value?,ε>, $\mathtt{v}_{a2}$ =<arg!,ε,value?>,
$\mathtt{v}_{s3}$ =<ack?,service!,ε>, $\mathtt{v}_{a3}$ =<ack?,ε,service!>.

Example 5 (Connected vs non connected modes). Suppose a client/server system where the client `C` sends its *id* only once at login time, while the server `S` requires an identification every time the client does a request. Here we have:

```
C[i]=log!.Logged, with
Logged[f]=req!.ack?.Logged, and
S[i,f]=log?.req?.ack!.S
```

The regex describing the adaptation required is now $\mathtt{v}_0.\mathtt{v}_2.\mathtt{v}_3.(\mathtt{v}_1.\mathtt{v}_2.\mathtt{v}_3)*$ with $\mathtt{v}_0$ =<log!,log?>, $\mathtt{v}_1$ =< ε,log?>, $\mathtt{v}_2$ =<req!,req?>, $\mathtt{v}_3$ =<ack?,ack!>.

4.2 Behavioural Adaptation Without Reordering

To be able to update our algorithms for using our new regex mappings[2], we first define how to obtain an LTS from them. This corresponds to the well-known problem of obtaining an automaton which recognizes the language of a regex [10]. The only difference is that the atoms of our regex are vectors and not elements of basic alphabets. Instead of using a regex, one may also use directly the LTS that derives from such regex, (*i.e.*, an LTS where the alphabet corresponds to vectors).

We then modify the synchronous vector product to take a regex LTS in place of the vector argument.

Definition 9 (Synchronous Vector Product (with regex LTS)). *The* synchronous vector product (with regex LTS) *of* n *LTS* $L_i = (A_i, S_i, I_i, F_i, T_i)$, $i \in 1..n$ *with a regex LTS* $L_R = (A_R, S_R, I_R, F_R, T_R)$, *is the LTS* (A, S, I, F, T) *such that:*

- $A \subseteq A_R \times \Pi_{i \in 1..n} A_i$, $S \subseteq S_R \times \Pi_{i \in 1..n} S_i$, $I = (I_R, I_1, \ldots, I_n)$,
- $F \subseteq \{(s_r, s_1, \ldots, s_n) \in S \mid s_r \in F_R \wedge \bigwedge_{i \in 1..n} s_i \in F_i\}$,
- T *is defined using the following rule:*
  $((s_r, s_1, \ldots, s_n), (l_r, l_1, \ldots, l_n), (s'_r, s'_1, \ldots, s'_n)) \in T$ *and* $(s'_r, s'_1, \ldots, s'_n) \in S$ *if*
  $\exists (s_r, s_1, \ldots, s_n) \in S$ *and* $\exists v = (s_r, (l_{r_1}, \ldots, l_{r_n}), s'_r) \in T_R$ *with,*
  $\forall l_{r_i}\ s'_i = s_i$ *if* $l_{r_i} = \varepsilon$ *and* $\exists (s_i, l_{r_i}, s'_i) \in T_i$ *otherwise.*

To apply the Section 3.2 algorithm we just have now to *discard* the first element of the product components, that is, from the LTS $L = (A, S, I, F, T)$ obtain the LTS $L' = \texttt{proj}(L) = (A', S', I', F', T')$ such that $\forall X \in \{A, S, I, F\}$ $X' = \{\texttt{cdr}(x) \mid x \in X\}$ and $T' = \{(\texttt{cdr}(s), \texttt{cdr}(l), \texttt{cdr}(s')) \mid (s, l, s') \in T\}$ with $\texttt{cdr}((x_0, x_1, \ldots, x_n)) = (x_1, \ldots, x_n)$.

We may now modify the algorithm for behavioural mismatching without reodering as presented in Section 3.2. The new algorithm takes as input the Id indexed set of components LTSs L_i of the system and a mapping which is a regex R (for the set of LTSs). We just have to replace step 1 in this algorithm by:

1. compute the LTS L_R for the regex R
2. compute the product $P_R = (A_{P_R}, S_{P_R}, I_{P_R}, F_{P_R}, T_{P_R}) = \Pi(L_R, L_i)$
3. compute $P = \texttt{proj}(P_R)$

Its complexity is $\mathbf{O}(|S|^{n+1})$ where S is the largest set of states.

4.3 Behavioural Adaptation with Reordering

Our algorithm for behavioural adaptation with reordering can also be adapted to deal with regex.

[2] Note that our new algorithms would apply to the vector mappings we have defined in the previous section, just taking the set $V = \{v_i\}$ of vectors as the regex $(v_1 + v_2 + \ldots + v_n)*$.

1. compute the LTS $L_R = (A_R, S_R, I_R, F_R, T_R)$ for the regex R.
2. build the Petri net encoding for the problem as presented in section 3.3, replacing part 6 with:
   - for each state s_R in S_R, add a place `ControlR-s_R`
   - put a token in place `ControlR-I_R`
   - for each transition $t_R = (s_R, (l_1, \ldots, l_n), s'_R)$ in T_R:
     - add a transition with label `tau`, one arc from place `ControlR-s_R` to the transition and one arc from the transition to place `ControlR-s'_R`
     - for each l_i which has the form $a!$, add one arc from place `Rec-a` to the transition
     - for each l_i which has the form $a?$, add one arc from the transition to place `Em-a`
3. in the building of `accept` transitions, add F_R to the F_i taken into account (final states now correspond to acceptance states of the regex LTS).

The rest of the algorithm (computing marking or coverability graph, and reducing them) is the same. Similarly to Section 3.3, this algorithm is exponential.

4.4 Application

We here develop Example 4 above, following our behavioural adaptation technique.

Example 6 (Example 4 developed). First note that, as explained before, we rename arguments to avoid name clash. We have:

```
C[i] = c:end![f] + c:req!.c:arg!.c:ack?.C,
S[i,f] = s:value?.s:query?.s:service!.S, and
A[i,f] = a:value?.a:query?.a:service!.A.
```

To express that the client alternatively uses the two servers we may use the following regex: $R_1 = (\mathtt{v}_{s1}.\mathtt{v}_{s2}.\mathtt{v}_{s3}.\mathtt{v}_{a1}.\mathtt{v}_{a2}.\mathtt{v}_{a3})^*.\mathtt{v}_{\text{end}}$ with:

$\mathtt{v}_{s1}$ =<c:req!,s:query?,ε>, $\mathtt{v}_{a1}$ =<c:req!,ε,a:query?>,
$\mathtt{v}_{s2}$ =<c:arg!,s:value?,ε>, $\mathtt{v}_{a2}$ =<c:arg!,ε,a:value?>,
$\mathtt{v}_{s3}$ =<c:ack?,s:service!,ε>, $\mathtt{v}_{a3}$ =<c:ack?,ε,a:service!>,
$\mathtt{v}_{end}$ =<c:end!,ε,ε>

Note that this mapping is probably overspecified, since it imposes a strict alternation between servers. Instead, one may choose to authorize the client to access any server it wants. Then, the mapping becomes:

$$R_2 = (\mathtt{v}_{s1}.\mathtt{v}_{s2}.\mathtt{v}_{s3} + \mathtt{v}_{a1}.\mathtt{v}_{a2}.\mathtt{v}_{a3})^*.\mathtt{v}_{\text{end}}$$

We have run both examples and obtained (after reduction) the adaptors in Fig. 6 (left for R_1, and right for R_2.) Note that applying step 2 of the algorithm presented in Section 3.2, the state `1` and the corresponding transition are removed for R_1. Both adapters solve the existing mismatch, making the system deadlock-free.

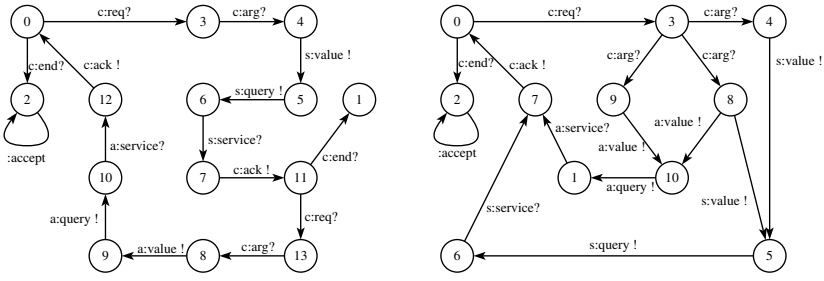

Fig. 6. Adaptors obtained for the alternating client/server system

5 Related Work

For a thorough review of the state of the art in Software Adaptation, we refer to [8]. Here, we will mention only a few works, those more closely related to our proposal.

As said in the introduction, the need for adaptation may occur at any of the levels of interoperability described, while currently available component platforms address software adaptation only at the signature level. Hence, most of the recent proposals for adaptation of software have jumped from the signature level to the specification and analysis of behavioural interfaces, promoting the use of BIDLs for describing component protocols.

The foundation for behavioural adaptation was set by Yellin and Strom. In their seminal paper [21], they introduced formally the notion of *adaptor* as a software entity capable of enabling the interoperation of two components with mismatching behaviour. They used finite state machines to specify component interactive behaviour, to define a relation of compatibility, and to address the task of (semi-)automatic adaptor generation.

More recently, in [18], the authors present an adaptation approach as a solution to particular synchronization problems between concurrent components, for instance one component uses or is accessed by two other components. This approach is based on algorithms close to the synchronous products we use in this paper. Moreover, they can solve protocol incompatibilities enabling one of the involved component to perform several actions before or after several synchronizations with its partners. In comparison, our proposal is more general and based on a rich notation to deal with possibly complex adaptation scenarios, whereas their approach works out only precise situations in which mismatch may happen, without using any mapping language for adaptor specification.

Taking Yellin and Strom's proposal [21] as a starting point, the work of Brogi and collaborators (BBCP) [4, 5] presents a methodology for behavioural adaptation. In their proposal, component behaviour is specified using a process algebra —a subset of the π-calculus—, where service offering/invocation is represented by input/output actions in the calculus, respectively. The starting point of the adaptation process is a mapping that states correspondences between services of

the components being adapted. This mapping can be considered as an abstract specification of the required adaptor. Then, an adaptor generation algorithm refines the specification given by the mapping into a concrete adaptor implementation, taking also into account the behavioural interfaces of the components, which ensures correct interaction between them according to the mapping. The adaptor is able to accommodate not only syntactical mismatch between service names, but also the interaction protocols that the components follow (*i.e.*, the partial ordering in which services are offered/invoked).

Another interesting proposal in this field is that of Inverardi and Tivoli (IT) [11]. Starting from the specification with MSCs of the components to be assembled and of the properties that the resulting system should verify (liveness and safety properties expressed as specific processes), they automatically derive the adaptor glue code for the set of components in order to obtain a property-satisfying system. The IT proposal has been extended in [12] with the use of temporal logic; coordination policies are expressed as LTL properties, and then translated into Büchi automata.

Our approach addresses system-wide adaptation (*i.e.*, differently from BBCP, it may involve more than two components). It is based on LTS descriptions of component behaviour, instead of process algebra as in BBCP. However, we may also describe behaviours by means of a simple process algebra, and use its operational semantics to derivate LTSs from it. Differently from IT, we use synchronous vectors for adaptor specification, playing a similar function than the mappings rules in BBCP. With that, we are able to perform adaptation of incompatible events.

With respect to behavioural adaptation, our approach can be considered as both generative and restrictive [8], since we address behavioural adaptation by enabling message reordering (as in BBCP), while we also remove incorrect behaviour (as in IT). Similarly to both approaches, our main goal is to ensure deadlock freedom. However, more complex adaptation policies and properties can be specified by means of regular expressions. Indeed, the most relevant achievement of our proposal is this use of regular expressions for imposing additional properties over mappings. In fact, the semantics of BBCP mappings can be expressed by combining their different rules (in our case, vectors) in a regular expression by

Table 1. Comparison of Adaptation approaches

criteria	IT	BBCP	our proposal
behavioural descriptions	automata	proc. algebra	LTS or proc. algebra
properties	no deadlock, LTL properties	no deadlock —	no deadlock regular expressions
mappings/adaptor abstraction	yes	yes	yes
name mismatch	no	yes	yes
data types	no	yes	no
message reordering	no	yes	yes
system-wide adaptation	yes	no	yes

means of the choice (+) operator. On the contrary, our regex are much more expressive, solving the problem of BBCP underspecified mappings [4], and allowing to take into account a new class of adaptation problems.

In Table 1 we give a synthesis of the features of our approach compared to IT and BBCP.

6 Conclusion

Software Adaptation has become a crucial issue for the development of a real market of components enhancing software reuse, especially when dealing with legacy systems. Recent research work in this field —in particular that of BBCP and IT [4, 5, 11, 12]— has addressed several problems related to signature and behavioural mismatch. In this paper, we have shown our proposal for software adaptation based on a notation, namely regular expressions of synchronous vectors, and equipped with algorithms and tools. It builds on BBCP and IT previous works, overcoming some of their limitations, and making a significant advance to solve some of the pending issues.

There are still some open issues in our proposal, deserving future work. First, and differently from BBCP, we do not deal with data types, nor with one-to-many correspondences between services. Taking data into account would require more expressive models than LTSs, such as Symbolic Transition Systems (STSs) [14]. This is a perspective for our work, since it allows the description of the data involved in the operations within the protocol without suffering from the state explosion problem that usually occurs in process algebraic approaches.

With respect to one-to-many correspondences between services (one of the strong points in favour of the BBCP proposal), we intend to explore how regular expressions can be used for that purpose. More expressive models for mappings, such as non-regular protocols [19], could also be extended to vectors in order to get a bigger class of properties expressible at the adaptor level (*e.g.*, load-balancing adaptation of the access of clients to servers).

Finally, we intend to implement our adaptation algorithms in ETS, an Eclipse plug-in that we have developed for the experimentation over LTS and STS.

Acknowledgements. The authors thank Bernard Berthomieu, Frédéric Lang, and Massimo Tivoli for their interesting comments and fruitful discussions.

References

1. T. Andrews et al. *Business Process Execution Language for Web Services (WS-BPEL).* BEA Systems, IBM, Microsoft, SAP AG, and Siebel Systems, Feb. 2005.
2. A. Arnold. *Finite Transition Systems.* International Series in Computer Science. Prentice-Hall, 1994.
3. B. Berthomieu, P.-O. Ribet, and F. Vernadat. The Tool TINA – Construction of Abstract State Spaces for Petri Nets and Time Petri Nets. *International Journal of Production Research*, 42(14), 2004.

4. A. Bracciali, A. Brogi, and C. Canal. A Formal Approach to Component Adaptation. *Journal of Systems and Software*, 74(1):45–54, 2005.
5. A. Brogi, C. Canal, and E. Pimentel. Component Adaptation Through Flexible Subservicing. *Science of Computer Programming*, 2006. To appear. A previous version of this work was published as *Soft Component Adaptation*, ENTCS 85(3), Elsevier, 2004.
6. A. W. Brown and K. C. Wallnau. The Current State of CBSE. *IEEE Software*, 15(5):37–47, 1998.
7. C. Canal, J. M. Murillo, and P. Poizat. Coordination and Adaptation Techniques for Software Entities. In *ECOOP 2004 Workshop Reader*, volume 3344 of *Lecture Notes in Computer Science*, pages 133–147. Springer, 2004.
8. C. Canal, J. M. Murillo, and P. Poizat. Software Adaptation. *L'Objet. Special Issue on Coordination and Adaptation Techniques*, 12(1):9–31, 2006.
9. H. Garavel, F. Lang, and R. Mateescu. An Overview of CADP 2001. *EASST Newsletter*, 4:13–24, 2002.
10. J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison Wesley, 1979.
11. P. Inverardi and M. Tivoli. Deadlock Free Software Architectures for COM/DCOM Applications. *Journal of Systems and Software*, 65(3):173–183, 2003.
12. P. Inverardi and M. Tivoli. Software Architecture for Correct Components Assembly. In *Formal Methods for Software Architectures*, volume 2804 of *Lecture Notes in Computer Science*, pages 92–121. Springer, 2003.
13. F. Lang. Exp.Open 2.0: A Flexible Tool Integrating Partial Order, Compositional, and On-The-Fly Verification Methods. In *Integrated Formal Methods (IFM'2005)*, volume 3771 of *Lecture Notes in Computer Science*, pages 70–88. Springer, 2005.
14. O. Maréchal, P. Poizat, and J.-C. Royer. Checking Asynchronously Communicating Components using Symbolic Transition Systems. In *Proc. of the International Symposium on Distributed Objects and Applications (DOA'2004)*, volume 3291 of *Lecture Notes in Computer Science*, pages 1502–1519. Springer, 2004.
15. T. Murata. Petri Nets: Properties, Analysis and Applications. *Proceedings of the IEEE*, 77(4):541–580, 1989.
16. O. Nierstrasz and T. D. Meijler. Research Directions in Software Composition. *ACM Computing Surveys*, 27(2):262–264, 1995.
17. C. Rackoff. The Covering and Boundedness Problems for Vector Addition Systems. *Theoretical Computer Science*, 6:223–231, 1978.
18. H. W. Schmidt and R. H. Reussner. Generating Adapters for Concurrent Component Protocol Synchronization. In *Proc. of the 5th Int. Conf. on Formal Methods for Open Object-Based Distributed Systems (FMOODS'02)*, pages 213–229. Kluwer Academic Publishers, 2002.
19. M. Südholt. A Model of Components with Non-regular Protocols. In *Proc. of Software Composition (SC'05)*, volume 3628 of *Lecture Notes in Computer Science*, pages 99–113. Springer, 2005.
20. C. Szyperski. *Component Software: Beyond Object-Oriented Programming.* Addison-Wesley, 1998.
21. D. M. Yellin and R. E. Strom. Protocol Specifications and Components Adaptors. *ACM Transactions on Programming Languages and Systems*, 19(2):292–333, 1997.