Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 229 (2009) 115–131

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Automatic Generation of Adaptation Contracts

José Antonio Martín[1 ,2]

Depto. de Lenguajes y Ciencias de la Computación
University of Málaga
Málaga, Spain

Ernesto Pimentel[3]

Depto. de Lenguajes y Ciencias de la Computación
University of Málaga
Málaga, Spain

Abstract

Software development based on the composition of black-box software like Web Services and Software Components is impeded by incompatibilities in their interfaces. Software adaptation has emerged as a solution to these incompatibilities by using processes in-the-middle (called adapters) that allow the correct interaction between the services. There are several approaches that focused on the automated generation of adapters guided by adaptation contracts which specify how the incompatibilities can be resolved. However, the generation of these contracts is not automated and most existing approaches require these contracts to be specified by hand, which obliges the designer to know all the service details. In this paper, we propose an approach to automatically generate adaptation contracts from the behavioral description of the services. These contracts overcome incompatibilities at signature and behavioral levels. Finally we present our prototype tool that accepts as input the service behaviors written in abstract BPEL and generates adaptation contracts using a combination of an A* algorithm and an expert system.

Keywords: behavioral adaptation, adapter specification, expert system, A* algorithm.

1 Introduction

Application design using black-box software such as Web Services and Software Components has several advantages like greater productivity and software reusability. Nevertheless this design based on black-box software has to face an important

[1] We thank the helpful comments and revisions of this work done by Gwen Salaün. This project has been partially funded by the Spanish Ministry of Science and Education (project TIN2007-67134) and Junta de Andalucía (project P06-TIC-02250).

[2] Email: jamartin@lcc.uma.es

[3] Email: ernesto@lcc.uma.es

doi:10.1016/j.entcs.2009.06.032

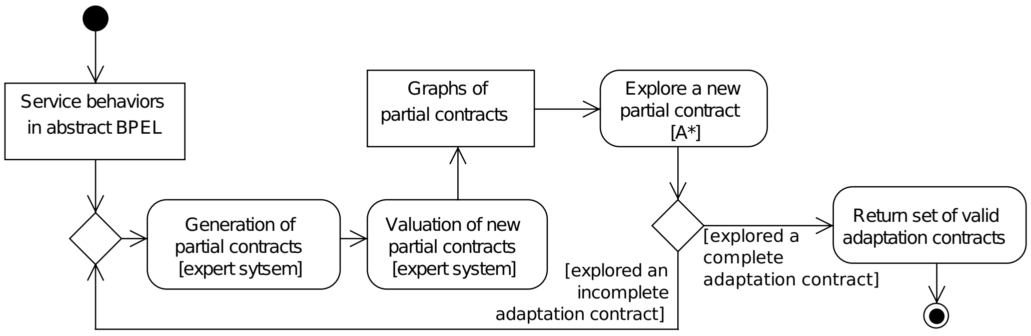

Service behaviors in abstract BPEL
Generation of partial contracts [expert sytsem]
Valuation of new partial contracts [expert system]
Graphs of partial contracts
Explore a new partial contract [A*]
[explored an incomplete adaptation contract]
[explored a complete adaptation contract]
Return set of valid adaptation contracts

Figure 1. Combination of expert system and A*

issue: the adaptation of components/services[4] with mismatches at signature and behavior levels [4]. Interface Description Languages (like CORBA and WSDL for components and services, respectively) allow the composition of software written in different languages but, even though IDLs help solve the language barrier, they do not address behavioral incompatibilities. This paper focuses on the adaptation of both signature and behavior inconsistencies.

Most of the time, services cannot be reused as they are because interactions among them would lead to an erroneous execution, namely a mismatch. Formally, cases of mismatch lead the whole system into deadlock states. In practice, mismatch situations may be caused by message names which do not correspond (regular use of the services makes them interact on the same names of messages), or by the order of messages not being respected, or a message required by one service not being provided by its partner. One possible solution to overcome such mismatches is using *adapters* as services "in-the-middle" capable of mediating between the involved parties. Adapters can be seen as Web service *orchestrators* which intercept client requests and forward them to the services while preserving a deadlock-free composition. In this way, adapters serve as a service replacement which properly support the interface expected by the clients.

Brogi, Bracciali, Canal and Pimentel [6,7] developed a formal methodology aimed at automatically deriving adapters from the interfaces (including the behavioral description) of the services. This methodology is based on the initial agreement between the parts involved about an abstract adaptation contract. This contract contains a mapping between the operations of the services in such a way that, when the adapter applies these correspondences, all the services cooperate properly and they end up in a final state. However, no insight was given about how this contract is constructed and it is assumed to be handmade. This is an error-prone task which obliges the designer to have a full understanding of all the service details. In this work we introduce an approach which addresses this issue.

We propose to generate contracts incrementally. Step by step, we explore the behavior of the services adding the messages found to the contract in all possible

[4] In the sequel, we use *service* as general term covering both Software Components and Web Services, i.e., a software entity to be composed within a system.

ways. An exhaustive exploration would lead to an explosion of partial contracts so we guide the search with a heuristic to restrict the number of contracts to explore. The exploration is made by an informed-search algorithm (A* [17]) whereas the contract validation is made by an expert system [11] (Figure 1).

The rest of the paper is organised as follows. In section 2 we introduce the subset of abstract BPEL used to describe service interfaces. We introduce a simplified example of a file exchange system in section 3 which will be used throughout the paper to illustrate the contract generation process. The notation for abstract contracts used throughout the concepts and ideas applied in the development of the proposal are described in section 4. Thereafter, in section 5, we explore the different parts of our approach and further details of the process. In section 6 we reference some related work and, finally, in section 7, we present future extensions and some conclusions.

2 Behavioral Interfaces in Abstract BPEL

We propose a subset of abstract BPEL [1] as the language to represent the service behaviors. This subset of abstract BPEL contains enough information to extract a behavioral interface that can be represented by *Finite State Machines* (FSMs). FSMs will allow us to graphically represent the behaviors in a more concise manner than BPEL and to define the operational semantics of the extracted model.

Definition 2.1 A *Finite State Machine* (FSM) is a quintuple $\langle S,\ L,\ s_0,\ F,\ \rightarrow\rangle$ where S is a set of *states*, L is a set of *labels*, $s_0 \in S$ is the initial state, $F \subseteq S$ is a set of final states, and $\rightarrow \subseteq S \times L \times S$ is a *transition relation* notated as $s_1 \xrightarrow{l} s_2$.

The states of an FSM are represented in terms of the following calculus:

$$t ::= \quad 0 \quad | \quad \alpha.t \quad | \quad \sum_i \tau.a_i!\bar{v}_i.t_i \quad | \quad \sum_i a_i?\bar{v}_i.t_i \quad | \quad t\,|\,t \quad | \quad P(\bar{v})$$

$$\alpha ::= \quad a!\bar{v} \quad | \quad a?\bar{v} \quad | \quad \tau$$

where terms can contain *silent* (τ) and *communicative actions* ($CAct = \{\ a?\bar{v},\ a!\bar{v} \mid a \in MessageNames,\ \bar{v} \in Args^n\}$). Communicative actions are represented by the name of the messages, an abstraction of their arguments between parenthesis ($\bar{v}$ being a list of arguments), and whether these messages are provided/accepted (?) or required/invoked (!). $L = \{\tau\} \cup CAct$. $P(\bar{v})$ corresponds to the instantiation of a procedure defined as $P(\bar{x}) \Leftarrow t$.

We use an operational semantics very similar to CCS [13] (see Figure 2). There is no value passing. We do not consider actual values but an abstraction of the arguments (i.e., argument names or data types). Therefore, messages are synchronized when they have the same name and arguments. There are several languages with their own semantics in the literature to describe Web service orchestration [5,10,18]. However, we have chosen this formalization because it is simple and abstract enough to cover our needs for behavioral matching.

$$\frac{}{a?\bar{v}.t \xrightarrow{a?\bar{v}} t}\ \text{(INPUT)} \qquad \frac{}{a!\bar{v}.t \xrightarrow{a!\bar{v}} t}\ \text{(OUTPUT)}$$

$$\frac{t \xrightarrow{l} t'}{t + u \xrightarrow{l} t'}\ \text{(SUM1)} \qquad \frac{u \xrightarrow{l} u'}{t + u \xrightarrow{l} u'}\ \text{(SUM2)}$$

$$\frac{t \xrightarrow{l} t'}{t \mid u \xrightarrow{l} t' \mid u}\ \text{(PAR1)} \qquad \frac{u \xrightarrow{l} u'}{t \mid u \xrightarrow{l} t \mid u'}\ \text{(PAR2)}$$

$$\frac{}{\tau.t \xrightarrow{\tau} t}\ \text{(TAU)} \qquad \frac{t \xrightarrow{l} t'}{t[f] \xrightarrow{f(l)} t'[f]}\ \text{(REN)}$$

$$\frac{t \xrightarrow{a?\bar{v}} t' \quad u \xrightarrow{a!\bar{v}} u'}{t \mid u \xrightarrow{\tau} t' \mid u'}\ \text{(COM)} \qquad \frac{t[\bar{v}/\bar{x}] \xrightarrow{l} t'}{P(\bar{v}) \xrightarrow{l} t'}\ P(\bar{x}) \Leftarrow t\ \text{(REC)}$$

Figure 2. Operational Semantics of the service interface model.

In our model, we focus on the following BPEL activities [5] (in bold):

`<invoke>` and `<reply>` activities are represented by $a!\bar{v}$ where the arguments ($\bar{v}$) are: either the single `inputVariable` attribute (or `variable` in `<reply>`), or several arguments contained inside a `<toParts>` element.

`<receive>` corresponds to $a?\bar{v}$ where $\bar{v}$ is handled in a similar way as the `<reply>` activity.

`<sequence>` is modelled using the period operator.

`<pick>` corresponds to $\sum_i a_i?\bar{v}_i.t_i$ where every $a_i?\bar{v}_i$ represents a different `<onMessage>` element.

`<if>` activities are expressed by $\sum_i \tau.a_i!\bar{v}_i.t_i$. Conditional expressions are abstracted by silent actions (τ) so the adaptation contract must assume that every execution branch is possible. The adapter must be notified about which branch has been selected in order to continue with the adaptation; therefore, every `<if>` branch has to start with a different `<invoke>` activity.

`<while>` and `<forEach>`. Because of the critical role played by the condition of these activities we model them as `<pick>` or `<if>` activities depending on whether the decision is made locally or on reception of a particular message. The branches of these activities are allowed to be loops, therefore we distinguish between *pick-loops* and *if-loops*.

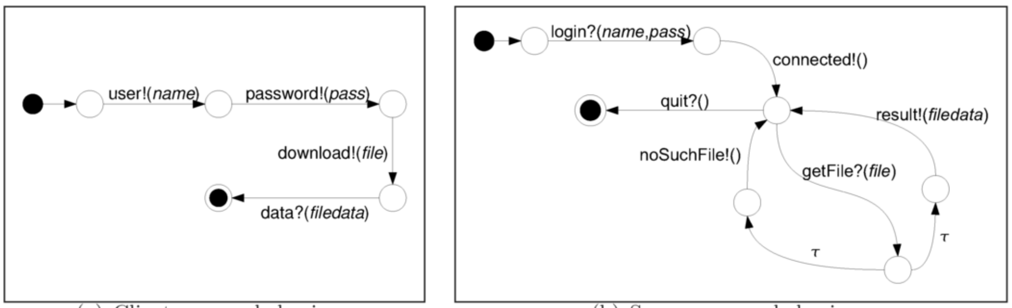
[5] See [1] for a complete description of the BPEL language, and Listing 1 for an example of these activities.

3 Motivating Example: File Exchange System.

We now introduce a case study that we will use throughout the paper to illustrate our approach. It consists of a file exchange system composed of a client and a server, but these were built in different contexts so they have mismatches in their signature and behavior. We provide the abstract BPEL code of the server[6] in Listing 1. As far as we focus on the adaptation between two parties, our BPEL code assumes that every message is received/sent from/to the other party instead of using `partnerLinks` and `portTypes` to define the particular source or destination.

Listing 1 Abstract BPEL code of the server process.

```
<process name=“server” ><sequence>
    <receive operation=“login”><fromParts >
        <fromPart part=“name” /><fromPart part=“pass” />
    </ fromParts ></ receive >
    <invoke operation=“connected” />
    <while ><condition>true</condition><! – – pick-loop ––>
        <pick >
            <onMessage operation=“quit”>
                <exit/>
            </ onMessage >
            <onMessage operation=“getFile” variable=“file”>
                <if ><condition opaque=“yes”>
                    <invoke operation=“result” inputVariable=“filedata”/>
                <else>
                    <invoke operation=“noSuchFile”/>
                </ else></if>
</ onMessage></ pick></ while ></ sequence ></ process >
```



(a) Client process behavior. (b) Server process behavior.

Figure 3. Example processes.

Figure 3 contains the FSMs of the client and server processes. The *server* (Figure 3(b)) accepts a connection given a user name and a password (`login?`)

[6] The abstract BPEL code of the client is a single sequence of activities.

```
c0 = [ user!(name), password!(pass)  ◇  login?(name, pass);   // (m1)
                                     ◇  connected!();         // (m2)
                      download!(file)  ◇  getFile?(file);       // (m3)
                      data?(filedata)  ◇  result!(filedata);    // (m4)
                      data?(filedata)  ◇  noSuchFile!();        // (m5)
                                     ◇  quit?(); ]            // (m6)
```

Table 1
Adaptation contract for the client (Figure 3(a)) and the server (Figure 3(b)).

and it confirms that the user has logged in (`connected!`). The user may perform several requests in a single session (`getFile?`) and every request has its single response. This response can be either the requested file (`result!`) or a notification that the requested file does not exist (`noSuchFile!`). Finally, when the client does not need more files, it leaves the session (`quit?`) and ends.

On the left-hand side, the *client* (Figure 3(a)) was designed with fewer transitions than the server. Name mismatches occur in every message and, although the client has similar request and data retrieval methods (`download!` and `data?`, respectively), it fails to receive the log-in confirmation (`connected!`) and the notification of `noSuchFile!` from the server. The client has the log-in request split into two actions (`user!` and `password!`) where the parameters *name* and *pass* satisfy their counterparts in the `login?` action of the server. The client also fails to call the `quit` operation of the server.

It is obvious that, even though these services cannot interact properly, they could be adapted to cooperate in most cases. In order to achieve this goal we must obtain mappings between the operations with signature incompatibilities but we must also merge messages (the log-in requests), and include the missing operations (`connected` and `quit`) in such a way that both services end up in a final state.

4 A Notation for Adaptation Contracts

Brogi *et al.* [6,7] introduced a simple, high-level notation for describing the correspondences between the transitions of two processes being adapted. This notation is used to represent the *abstract adaptation contract* by a set of *mappings* between the operations of two services (that we will refer to as *left* and *right* services). Here, we will briefly present this notation which will be used to represent the generated contracts.

Definition 4.1 A *mapping* $(la_1, \dots la_L \diamond ra_1, \dots ra_R)$ is composed of two sequences of actions such that $la_i, ra_j \in CAct$ and they belong to the *left* and *right* services respectively. The set of mappings will be denoted by M.

Definition 4.2 An *adaptation contract* c is an array of mappings $[m_1, \dots m_n]$. We will denote by $(c \in)C$ the set of contracts.

The methodology presented in [6,7] does not require a specific mappings order within a contract. However, the contract generation process needs to know which is the last generated mapping, so we structured them in an array.

Now we will explain what a programmer would do to design an abstract contract (Table 1) for the example in section 3. The programmer knows how a download session must proceed (*its behavior*) and the correlation among *the arguments*. With all that knowledge, it is common sense that `login` must match the combination of `user` and `password`. It is so because they are at the beginning of their respective services so they will be called or received at the same stage of the communication. Also, as far as `login` requires the arguments of the other two actions, they will all be merged in the same mapping (`m1`). The mappings `m3` and `m4` are simpler versions of the same case. These two are perfect mappings because they directly adapt a single call with its reception and all the arguments are satisfied. They only overcome signature mismatches.

The mappings `m2` and `m5` allow transitions unsupported by one of the services to be ignored and proceed with the communication. The mapping `m5` requires additional consideration because the argument `filedata` is not provided and it is required to reach a final state. Finally, we must call the `quit` method when the transaction ends (accomplished by `m6`). Notice that the execution of `m6` is not triggered by any other interaction so the adapter which complies with this contract must trigger `m6` based on its knowledge of the process behaviors.

Our approach takes advantage of the following information in order to achieve similar reasoning abilities to those stated above:

Behaviors We traverse the execution of the services using their behavioral interfaces. We can analyze the sequence of actions taking place and evaluate the most compatible mappings for those sequences. Some characteristics to be measured are the compatibility in the communication (invoke and receive pairs), well balanced mappings with similar number of actions in both sides, and the satisfaction of the arguments. These concepts will be explained in detail in subsection 5.1.

Arguments We make our best effort to satisfy the arguments required by one side of the mapping with those provided in the other side of the mapping. It is still possible to generate contracts where the reception of the argument is in one mapping and it is used in a different mapping but this would require the adapter to keep track of the arguments received. Therefore, we promote adaptation contracts where no argument memory is needed for the sake of scalability. Anyway, if there were no other alternatives, the actions with these arguments would be split into different mappings.

5 Generating Adaptation Contracts

We tackle the generation of adaptation contracts by adding, step by step, new mappings to an empty contract. During this process, we may only modify the last mapping or append a new one at the end (see Figure 4). The service behaviors are traversed and the actions found are introduced into the contracts (underlined actions) in all possible combinations. In this way, we iteratively create more complete contracts.

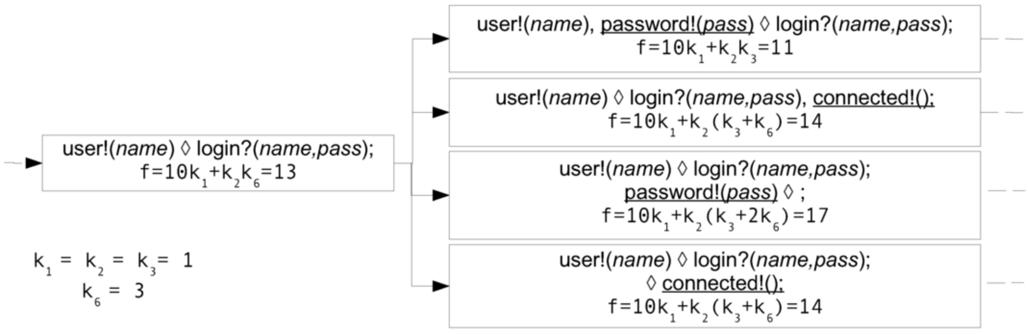

user!(name) ◊ login?(name,pass);
f=10k1+k2k6=13
k1 = k2 = k3= 1
k6 = 3
user!(name), password!(pass) ◊ login?(name,pass);
f=10k1+k2k3=11
user!(name) ◊ login?(name,pass), connected!();
f=10k1+k2(k3+k6)=14
user!(name) ◊ login?(name,pass);
password!(pass) ◊ ;
f=10k1+k2(k3+2k6)=17
user!(name) ◊ login?(name,pass);
◊ connected!();
f=10k1+k2(k3+k6)=14

Figure 4. Part of a graph of incremental contracts [7] .

In the example (Figure 3), the arguments are already matched between the services. This is a requirement of our approach, i.e., both services must be defined with the same set of argument names. One way to achieve this match is to represent the arguments by their *data types*. In this case, our approach will promote mappings which adapt messages with the same data types.

5.1 Graph Search with A*

The evaluation of all possible combinations of the service behaviors would lead us to an explosion of states (partial contracts). Therefore, the search through those states must be guided to greatly reduce the number of explored states. The concepts stated at the end of section 4 can be translated into a heuristic to guide the search using an informed-search algorithm (specifically A* [17]). Informed-search algorithms require a *cost* and a *heuristic* function. The former is the cost to reach a particular point of the search while the latter is a guess of how much further the solution might be from that point. During the incremental construction of the contract, the *cost* of a contract is how many mismatches have been assumed in conjunction with how many partial contracts are in the path to that contract. The solution to this search will be a complete adaptation contract with the lowest cost and heuristic. We will design the heuristic and cost functions based on the mappings which constitute the contract.

Definition 5.1 The *valuation* v of a mapping $m = (la_1, \cdots, la_L \Diamond ra_1, \cdots, ra_R)$ is defined as follows:

$$v(m) = k_3 \left| \sum_{i=1}^{L} rec(la_i) - \sum_{i=1}^{R} rec(ra_i) \right| + k_3 \left| \sum_{i=1}^{L} sen(la_i) - \sum_{i=1}^{R} sen(ra_i) \right| \quad (1)$$
$$+ \, mindet(m) \quad (2)$$
$$+ \, k_6 * ins(m) \quad (3)$$

[7] The function f, which represents the penalization associated to the given contract, will be explained in detail in subsection 5.1.

where

$$mindet(m) = \begin{cases} k_4 * rec(la_1) & \text{if } R = 0 \wedge L > 0; \\ k_4 * rec(ra_1) & \text{if } L = 0 \wedge R > 0; \\ k_5 * rec(la_1) * rec(ra_1) & \text{if } L > 0 \wedge R > 0; \\ 0 & \text{otherwise} \end{cases} \qquad (mindet)$$

$$rec(x) = \begin{cases} 1 & \text{if } x = a?\bar{v}; \\ 0 & \text{otherwise} \end{cases} \qquad (rec)$$

$$sen(x) = 1 - rec(x) \qquad (sen)$$

The function $ins : M \to \mathbb{N}$ is defined in such a way that $ins(m)$ is the number of unsatisfied arguments in m, that is, the number of provided/required arguments in one side which are not required/provided by the other side. Positive constants k_3, k_4, k_5 and k_6 weigh the different valuation terms.

The purpose of the valuation of a mapping (v) is to represent how bad a single mapping is. The higher the mapping valuation, the worse for the adaptation. A perfect mapping should have a value of 0. The function v is informally explained as:

Balance: The first line (1) of the equation defining v includes a penalization because of an unbalanced mapping. If two services are directly adaptable, an ideal adaptation contract would contain a mapping for each pair of actions. Each of these mappings would contain a single action on each side (one per service) representing that these actions must be directly adapted. Therefore, the actions of the services are adapted one to one.

Mapping indeterminism: Line 2 stands for the penalization of mappings which starts with receive actions in both sides. This is so because the adapter should trigger those mappings by its own responsibility, without receiving any message from the services indicating such a thing. Nonetheless, in some cases, it is possible to know without ambiguity when such mappings should be triggered.

Satisfiability: Every argument sent should be used and every argument needed must be satisfied. We can achieve these objectives by promoting those mappings where all the arguments are used and satisfied. If all the arguments are satisfied in the same mapping (and not in subsequently fired mappings) no argument memory will be required in the adapter. The penalization for unsatisfied arguments in line 3 serves to correlate actions based on their arguments and it enhances the adapter efficiency.

There are constants to weigh *balance* (k_3), *mapping indeterminism* (k_4 and k_5) and *satisfiability* (k_6) according to our adaptation policy.

An adaptation contract can be indeterministic in two ways: when a mapping is not triggered by any message received (*mapping indeterminism*), and when the same sequence of messages triggers several mappings (*contract indeterminism*). In order to define the latter, we need to know when two sequences of messages are distinguishable by the adapter.

Definition 5.2 Two sequences of communicative actions $\bar{a} = a_1, \ldots, a_n$ and $\bar{a}' = a'_1, \ldots, a'_m$ are *distinguishable* if, and only if:

$$dist(\bar{a}, \bar{a}') \Leftrightarrow \exists j > 0 \mid \forall i,\ 0 < i < j,\ (a_i = a'_i),\ sen(a_i) = sen(a'_i) = 1 \text{ and:}$$
$$\begin{array}{lll} \text{i)} & \text{if } m < j \le n & \text{then } sen(a_j) = 1; \\ \text{ii)} & \text{if } n < j \le m & \text{then } sen(a'_j) = 1; \\ \text{iii)} & \text{if } j < n, m & \text{then } sen(a_j) + sen(a'_j) \ge 1 \text{ and } a_j \neq a'_j; \end{array} \quad (dist)$$

Informally, two sequences are distinguishable if they differ in at least one invoke action and if all the previous pairs of actions are the same and are invocations. This particular definition of *dist* requires a timeout in the adapter to distinguish between sequences where only one of the first different actions is an invoke operation. For instance, the sequences $a_1!\bar{v_1},\ a_2!\bar{v_2},\ \ldots$ and $a_1!\bar{v_1},\ a_3?\bar{v_3},\ \ldots$ require a timeout because, after receiving $a_1!$, we cannot know if we need to call $a_3?$ or if we must wait to receive $a_2!$. For this reason, we must delay (with a timeout) the invocation of $a_3?$ waiting for the possible reception of $a_2!$.

Definition 5.3 Two mappings (m and m') such as $m = (la_1, \ldots la_L \Diamond\ ra_1, \ldots ra_R)$ and $m' = (la'_1, \ldots la'_{L'} \Diamond\ ra'_1, \ldots ra'_{R'})$, are *ambiguous* if, and only if:

$$amb(m, m') \Leftrightarrow \neg dist(\bar{la}, \bar{la}'),\ \neg dist(\bar{ra}, \bar{ra}') \text{ and either:}$$
$$\begin{array}{ll} \text{i)} & L, L' > 0 \text{ and } la_1 = la'_1,\ sen(la_1) = sen(la'_1) = 1; \\ \text{ii)} & R, R' > 0 \text{ and } ra_1 = ra'_1,\ sen(ra_1) = sen(ra'_1) = 1; \end{array} \quad (amb)$$

Two mappings are considered ambiguous if they are triggered by the same sequence of invocations and their sides are not distinguishable.

Definition 5.4 *Contract indeterminism* is penalized by the function $cindet : C \rightarrow \mathbb{N}$, defined as:

$$cindet(c) = \begin{cases} k_7 & \text{if } \exists i, j,\ i \neq j \mid m_i, m_j \in c \text{ and } amb(m_i, m_j); \\ 0 & \text{otherwise} \end{cases} \quad (cindet)$$

We can define the heuristic and the cost of a contract depending on how bad its mappings are (v). As we saw in Figure 4, any child contract (right hand side) has one action more than its father. This action will be either joint with the last mapping of its father or it will create a new mapping. Therefore, only the last mapping can be modified so all the other mappings are immutable in the descendants of the contract. The value (v) of the last mapping belongs to the heuristic because of its dynamic nature while the values of the other mappings belong to the cost because they will not be changed.

We promote contracts with the lowest number of actions so every action included in a contract will increase the cost of that contract. Therefore, the number of remaining actions is a good estimation of the future cost of the final solution and it belongs to the heuristic.

Definition 5.5 The *heuristic* (h) and *cost* (g) functions ($h, g : C \rightarrow \mathbb{N}$) establish the decision criteria of the A* algorithm. Given a contract $c = [m_1, \ \dots m_n]$, h and g are defined as follows:

$$h(c) = k_2\,(cindet(c) + v(m_n)) + k_1 * max\,(0,\ N - n(c)) \qquad (h)$$

$$g(c) = k_1 n(c) + k_2 \sum_{i=1}^{n-1} v(m_i) \qquad (g)$$

where N is the number of communicative actions in the services, and $n : C \rightarrow \mathbb{N}$ the number of communicative actions in the given contract. Constants $k_1 > 0$ and $k_2 \geq 0$ adjust the importance given to the number of actions in the contract, and the weight of v and $cindet$, respectively.

In Figure 4, we can see that the most promising contract (the top child) will be selected for exploration because it has the lowest value of $f(c)\,(= g(c) + h(c))$. The only condition is that the *satisfiability of the arguments* must have a positive weight ($k_6 > 0$).

This methodology for the generation of partial contracts along with the costs and the heuristics fits in any *informed search algorithm*. We use an A* variation in order to perform and guide the search. We have modified the A* algorithm because we do not just need one A* search but several of them, that is the case of branches in the execution flow. When the service is able to receive several messages and it follows its execution based on the message received (i.e., `<pick>`), we can model those branches directly by different branches of the A* tree. This is so because of the crucial role played by the communication in the decision. It is completely different in local choices where the decision is made by the service without any communication (i.e., `<if>`). Hence, we have to create several new search trees, one per different choice. Finally, it should also be considered that these different trees will eventually collide (when the conditional behavior ends) and, therefore, we have to merge those partial contracts into a new complete one. We will not go into details but the creation and merging of these trees has a significant impact on the performance.

A* is an exhaustive search algorithm, i.e., it will explore every possible contract (ordered by their cost and heuristic) until it finds a solution. Therefore, as far as (i) it is fed with every possible combination which follows a given partial contract, (ii) the service behaviors have one or more reachable final states, and (iii) the cost and heuristic function avoid infinite paths since the cost is strictly increasing, the search algorithm will eventually find a deadlock-free solution. In the worst case, the A* algorithm has an exponential time and memory complexity.

If no better solution is found, our approach will generate a *trivial contract*, one that describes an adapter which accepts (and ignores) every message and calls every needed operation with made-up arguments. Therefore, no decision is made about whether the given services should be adapted or not. The heuristic and cost values (f) of the solution contracts are a good measure of the incompatibilities between the services but the adapter-derivator is the one that must finally decide whether it can implement or not the generated contracts.

5.2 The Expert System

It was a requirement that all the valuation criteria stated in subsection 5.1 could be easily replaced or modified. In this way we could test new valuation techniques, include semantic information, or customize our contract generation process to our particular needs or environment restrictions. This is achieved using an *expert system* [11].

The expert system is in charge of traversing the behaviors of the services to generate the different alternatives available from a given partial contract. It calculates their costs and heuristics based on the new included action and it feeds the A* algorithm with the generated graph. The search algorithm makes its choice and marks it to be further explored so the expert system starts again from the chosen partial contract (Figure 1). The expert system also recognizes when a generated contract is a possible solution by examining the traces which leaded to that partial contract. If both traces reach a final state and the contract contains all the required mappings, the contract is complete. Once the A* explores any of these complete contracts, the process ends returning that contract and all the other solutions with the same value of f. All of this functionality is achieved by 62 rules (Figure 5) which are executed every time the A* algorithm explores a new partial contract.

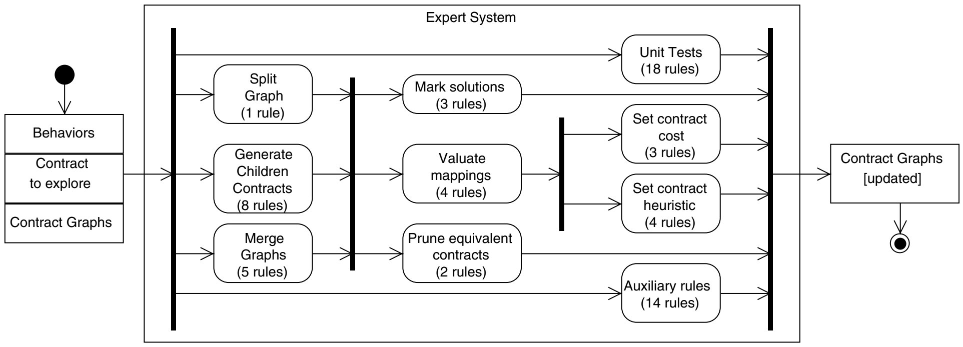

Expert System
Behaviors
Contract to explore
Contract Graphs
Split Graph (1 rule)
Generate Children Contracts (8 rules)
Merge Graphs (5 rules)
Mark solutions (3 rules)
Valuate mappings (4 rules)
Prune equivalent contracts (2 rules)
Unit Tests (18 rules)
Set contract cost (3 rules)
Set contract heuristic (4 rules)
Auxiliary rules (14 rules)
Contract Graphs [updated]

Figure 5. Activity diagram of the expert system rules. Every activity corresponds to one or more rules of the expert system.

Listing 2 contains one of the rules stated above, in particular the rule in charge of splitting the search graph when it finds conditional branches. The client and server processes are differentiated by their ?side. This rule is triggered when a contract is marked for further exploration by the A* algorithm (**childrenNeeded** TRUE) and the node to process is an ?ifActivity. Then it retracts the fact that the `<if>` must be processed and splits the search into as many graphs as conditional branches.

Listing 2 Rule which starts the search split.

```
(defrule split-graph
    (BehaviorNode (nodeType "IF") (OBJECT ?ifActivity))
    ?fact <- (activity-to-process (activity ?ifActivity) (side ?side)
        (contract ?contract) (behavior ?behavior))
    (Contract (OBJECT ?contract) (childrenNeeded TRUE))
    =>
    (retract ?fact)
    (foreach ?child ((?behavior getChildren ?ifActivity) toArray)
        (splitGraph ?ifActivity ?child ?side ?contract)))
```

5.3 Prototype Tool: Dinapter

We have implemented our proposal in a tool called Dinapter[8]. It takes as inputs the behaviors of the services to be adapted, described using abstract BPEL. Those behaviors are internally modeled into two directed graphs that will be explored during the automatic generation of the contracts. The output is a set of adaptation contracts expressed in the notation introduced in section 4.

	e01	s02	v03	e06	e10	e07	e04	e12	e02c	e16
Picks	1	1	2	2	2	0	3	2	1	3
Ifs	0	0	0	1	0	1	1	2	1	3
Loops	×	×	×	×	×	×	×	√	√	×
Client	3	6	5	5	9	2	5	6	4	7
Server	4	5	6	4	7	7	12	6	6	7
Mappings	21	41	52	34	76	45	95	31	57	87
Trees	1	1	1	19	1	50	43	74	155	310
Exp. trees	1	1	1	9	1	18	31	48	53	116
Cons.	31	79	120	82	191	180	341	206	440	681
Exp. cons.	10	25	32	38	43	100	142	142	258	365
Useless	0	0	0	0	0	0	2	2	2	0
Solutions	1	1	2	2	1	9	4	0(1)	4	1

Table 2
Some results obtained from Dinapter[9]

In Table 2 there are some data gathered from Dinapter run in several examples. The rows are as follows: the first (*Loops*) is the presence of loops in the example while *Picks* and *Ifs* are the number of event driven conditions (`<pick>`) and regular conditional behavior (`<if>`) in the services. The number of communicative actions (`<invoke>`, `<reply>`, `<receive>`, and `<onMessage>`) in the client and the server are

[8] Available at `http://sourceforge.net/projects/dinapter`.

[9] $k_1 = k_2 = k_3 = 1$; $k_4 = 0$; $k_5 = 50$; $k_6 = 3$; $k_7 = 100$

Client and *Server* respectively. The next column is the total number of generated *Mappings* followed by the number of search *Trees*. *Cons.* is the number of partial contracts. *Exp. trees* and *Exp. cons.* are the number of search trees and contracts explored before reaching a solution. *Useless* is the number of solutions that, in spite of being deadlock-free, adapt a branch of an event driven condition where no useful results are obtained from the adaptation (e.g., a client which only connects and disconnects without doing any computation). This happens because the heuristic and cost functions consider that it is better to connect and disconnect than to deal with the incompatibilities that would be found otherwise. The last row is the number of valid *Solutions* found.

Let us comment on two of these examples. The two processes in e12 are able to accept or reject the communication before performing their core functionality so, as their behaviors are quite incompatible, the first contract returned by Dinapter makes them refuse to communicate and end up in a final state. Nonetheless, if we execute another iteration of the process in the example e12, it returns a valid solution. The example e02c is the one described in section 3.

A remark from this table is the fact that the most relevant factor for the complexity of our approach is the number of nodes which alter the execution flow (`<pick>`, `<if>`, and loops) which is much more important than the number of transitions; another important point is that, if the parameters (k_i) are not adjusted in accordance with our adaptation policy (or the services have unsolvable incompatibilities), it might yield useless results as in the examples e02c, e04 and e12.

Another interesting point is the relevant role played by the A* algorithm and the underlying heuristic function which, even though there is a state explosion if the problem is difficult enough, it reduces the number of explored nodes to half of the nodes generated, approximately. As we stated at the end of subsection 5.1, the number of generated search trees (*Trees*) is proportional to the number of explored contracts (*Exp. Cons.*). Any enhancement in the heuristic and the procedure for generating and merging those trees will greatly improve the efficiency of our tool.

6 Related Work

Software adaptation is a very promising topic and it has been successfully applied to different implementation platforms such as BPEL [8] or Windows Workflow Foundation [10]. Several proposals [7,9,20] already focused on signature and behavioral adaptation. However, all these approaches do require a manual design of the adaptation contract which may be tricky when the service protocols are complicated. Our solution complements these works by generating adaptation contracts from behavioral descriptions of services, which makes the adaptation process completely automated.

Moser *et al.* [14] developed a platform (VieDAME) based on ActiveBPEL for the monitoring and service adaptation of BPEL processes. They dynamically replace services based on QoS in a non-intrusive manner using aspect oriented programming. They adapt services using *Transformers* but these transformers must be designed manually. Their work can be complemented by our approach by automatically generating these transformers.

As regards automatic generation of adaptation contract, Schmidt and Reussner [19] focused on the synchronization of two components accessing, or being accessed, by a third one. They introduced an algorithm based on synchronous product computation to semi-automatically solve missing message incompatibilities, but their approach fails to overcome signature mismatches and behavioral incompatibilities like message reordering or message splitting/merging. Autili *et al.* [3] proposed a methodology for the automatic synthesis of adapters considering as input behavioral descriptions of components and a specification of the interactions that must be enforced in the system. Then, their tool (Synthesis) generates composition code that exhibits only the specified interactions, and prunes those which lead to deadlocks. Similarly to [19], same names of messages are assumed and some behavioral mismatches cannot be solved (such as message splitting/merging). In addition, this approach relies on a high-level description of the composition goal, and therefore does not work without such specification.

Let us now mention two related works [8,15] that tackled Web Service adaptation. In the first one, Brogi and Popescu [8] outlined a methodology for the automated generation of adapters capable of solving behavioral mismatches between BPEL processes. In their adaptation methodology they use the YAWL workflow as an intermediate language. Once the adapter workflow is generated, they use lock analysis techniques to check if a full adapter has been generated or only a partial one (some interaction scenarios cannot be resolved). They solve message reordering incompatibilities but their approach fails with signature mismatches. In addition, even if we applied our approach to BPEL services as well, we want our approach to be more general by working on abstract descriptions of services that can be extracted from BPEL but also from other programming languages and platforms such as Windows Workflow Foundation [10].

Motahari Nezhad *et al.* [15] presented an approach for assisting the developer to adapt new versions of existing Web Services. In their approach, they use a schema matching tool called COMA++ [2] over the service WSDL signatures. Our approach has some similarities with their work (our heuristic plays a similar role as their *evidences*), and they introduce some interesting ideas about deadlock handling. However, although they are able to generate a mismatch tree that gathers all protocol mismatches, its resolution is not automatic.

7 Concluding Remarks

In this work, we have shown an approach for the automatic generation of adaptation contracts which overcomes signature and behavioral mismatches. The generated contracts successfully solve missing messages and they are able to merge and split messages depending on their arguments. There are several works in the literature [7,9,20] which use these contracts to automatically build behavioral adapters. Traditionally, these contracts were manually written and they required the designer to fully understand the details of the services involved. Our proposal complements these works by the automatic generation of these contracts and it is supported by a prototype tool we implemented.

As our approach traverses the service behaviors while generating the adaptation contract, we can infer the sequence in which the adapter will use those mappings. Future work for our approach is to compose a graph of how the mappings should be used by the adapter. This graph would ease the derivation of the adapter and reduce the mapping and contract indeterminisms.

The A* algorithm requires the heuristic function to be admissible and monotonous (because of the possible infinite space of partial contracts) for the solution to be optimal. The proposed heuristic function (h) may decrease drastically in the following steps and this causes the heuristic not to be admissible. This inconvenience can be controlled by the constants k_1 and k_2. With values of $k_2 \geq k_1$ we can promote a faster and narrower search assuming the risk of missing the best solution or, otherwise, we can force the tool to find the best solution generating more partial contracts with $k_2 \approx 0$. Other informed search algorithms can be used instead of A*.

Our work has been focused on contract generation between two services. Future work is to extend our approach to more expressive languages and semantics like STS [12], or SCC [5], which is focused on service orchestration and it supports explicit session and exception handling. In any case, this work shows the feasibility of the proposed approach. Regarding validation, another of our future works is to apply our tool results to other methodologies for adapter generation and to check their bisimilarity with adapters generated from hand-writen contracts.

Unlike most proposals for mapping generation [15] and schema matching [2], our approach relies only on the behaviors and the arguments of the services. However, our approach can be extended to include semantic and syntactic information to improve its heuristics. One way to achieve this could be to compare the semantics behind the operations and their arguments using Wordnet::Similarity [16].

The combination of heuristic, cost and A* quickly solves simple mismatches but, the bigger the incompatibilities are, it consumes more time exploring other ways to overcome them. This allows our approach to tackle different degrees of incompatibility but it wastes too much time when the incompatibilities are irremediable. One course of action would be to complement our proposal with an algorithm to automatically recognize these irremediable incompatibilities or an algorithm to cut service behaviors into smaller adaptable pieces.

References

[1] Andrews, T. et al., "Business Process Execution Language for Web Services (WSBPEL)," BEA Systems, IBM, Microsoft, SAP AG, and Siebel Systems (2005).

[2] Aumueller, D., H. Do, S. Massmann and E. Rahm, *Schema and ontology matching with COMA++*, in: *Proc. of SIGMOD'05* (2005), pp. 906–908.

[3] Autili, M., P. Inverardi, A. Navarra and M. Tivoli, *SYNTHESIS: A Tool for Automatically Assembling Correct and Distributed Component-Based Systems*, in: *Proc. of ICSE'07* (2007), pp. 784–787.

[4] Becker, S., A. Brogi, I. Gorton, S. Overhage, A. Romanovsky and M. Tivoli, *Towards an engineering approach to component adaptation*, in: S. B. . Heidelberg, editor, *Architecting Systems with Trustworthy Components*, LNCS **3938**, Springer, 2006 pp. 193–215.

[5] Boreale, M., R. Bruni, R. De Nicola, I. Lanese, M. Loreti, F. Martins, U. Montanari, A. Ravara, D. Sangiorgi, V. Vasconcelos and G. Zavattaro, "SCC: A Service Centered Calculus," LNCS **4184**, Springer, 2006 pp. 38–57.

[6] Bracciali, A., A. Brogi and C. Canal, *A formal approach to component adaptation*, The Journal of Systems & Software **74** (2005), pp. 45–54.

[7] Brogi, A., C. Canal and E. Pimentel, *On the semantics of software adaptation*, Science of Computer Programming **61** (2006), pp. 136–151.

[8] Brogi, A. and R. Popescu, *Automated Generation of BPEL Adapters*, in: *Proc. of ICSOC'06*, LNCS **4294** (2006), pp. 27–39.

[9] Canal, C., P. Poizat and G. Salaun, *Model-Based Adaptation of Behavioural Mismatching Components*, IEEE Trans. on Softw. Eng. (2008), to appear.

[10] Cubo, J., G. Salaün, C. Canal, E. Pimentel and P. Poizat, *A Model-Based Approach to the Verification and Adaptation of WF/.NET Components*, in: *Proc. of FACS'07*, ENTCS (2007), to appear.

[11] Friedman-Hill, E., "Jess in Action: Java Rule-Based Systems," Manning Publications Co., 2003.

[12] Ingolfsdottir, A. and H. Lin, *A symbolic approach to value-passing processes*, in: *Handbook of Process Algebra*, Elsevier, 2001 .

[13] Milner, R., "A Calculus of Communicating Systems," Springer, 1982.

[14] Moser, O., F. Rosenberg and S. Dustdar, *Non-Intrusive Monitoring and Adaptation for WS-BPEL*, in: *Proc. of WWW'08*, 2008, to appear.

[15] Motahari Nezhad, H. R., B. Benatallah, A. Martens, F. Curbera and F. Casati, *Semi-automated adaptation of service interactions*, in: *Proc. of WWW'07* (2007), pp. 993–1002.

[16] Pedersen, T., S. Patwardhan and J. Michelizzi, *Word-Net::Similarity - Measuring the relatedness of concepts*, in: *Proc. of Intelligent Systems Demonstrations, held in AAAI* (2004), pp. 267–270.

[17] Russel, S. and P. Norvig, "Artificial Intelligence: a Modern Approach," Prentice-Hall, 1995.

[18] Salaün, G., L. Bordeaux and M. Schaerf, *Describing and Reasoning on Web Services using Process Algebra*, International Journal of Business Process Integration and Management **1** (2006), pp. 116–128.

[19] Schmidt, H. and R. Reussner, *Generating adapters for concurrent component protocol synchronisation*, in: *Proc. of FMOODS'02* (2002), pp. 213–229.

[20] Yellin, D. and R. Strom, *Protocol Specifications and Component Adaptors*, ACM Trans. Program. Lang. Syst. **19** (1997), pp. 292–333.