HAL
open science

Semi-automatic Specification of Behavioural Service Adaptation Contracts

Javier Cámara, Jose Antonio Martin, Gwen Salaün, Carlos Canal, Ernesto Pimentel

To cite this version:

Javier Cámara, Jose Antonio Martin, Gwen Salaün, Carlos Canal, Ernesto Pimentel. Semi-automatic Specification of Behavioural Service Adaptation Contracts. 7th International Workshop on Formal Engineering approaches to Software Components and Architectures (FESCA'10), Mar 2010, Paphos, Cyprus. pp.19-34. inria-00539116

HAL Id: inria-00539116

https://inria.hal.science/inria-00539116v1

Submitted on 25 Nov 2010

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Semi-Automatic Specification of Behavioural Service Adaptation Contracts

Javier Cámara[a], José Antonio Martín[b], Gwen Salaün[c], Carlos Canal[b], Ernesto Pimentel[b]

[a] *INRIA Grenoble - Rhône-Alpes, France*
Email: Javier.Camara-Moreno@inrialpes.fr

[b] *Department of Computer Science, University of Málaga, Spain*
Email: {jamartin,canal,ernesto}@lcc.uma.es

[c] *Grenoble INP- INRIA Grenoble - LIG, France*
Email: Gwen.Salaun@inrialpes.fr

Abstract

An adaptation contract describes composition constraints and adaptation requirements among several services which were not initially built to interoperate with each other. The manual writing of this specification by a designer is a difficult and error-prone task, especially when services are reused taking their behavioural descriptions into account. In this paper, we present a semi-automatic approach to build adaptation contracts. To this purpose, we propose an adaptation contract design process supported by an interactive environment based on a graphical notation, and an engine capable of automatically generating contracts without any human intervention. We also present an experimental study that we carried out using the tool support that we implemented in order to evaluate our approach.

Keywords: Service, Composition, Adaptation, Behavioural Interface, Contract, CASE

1 Introduction

Building software systems as a combination of interacting entities aims at improving productivity since it enables the reuse of third-party, pre-existing software components or services which are selected and assembled to build a new system. However, one cannot expect any given service to perfectly match the needs of the new system or composition at hand, thereby its integration may require some adaptations in order to solve potential mismatch situations with the rest of the services. *Software adaptation* [20,9] is a hot topic in Software Engineering since it is the only way to compose non-intrusively black-box components or services with mismatching interfaces by automatically generating mediating *adaptor* components. These are automatically built from an abstract specification of how mismatches can be solved (*i.e.*, an *adaptation contract*).

Mismatches may appear at different interoperability levels that are usually distinguished in interface description languages [5]: signature level (operation names and types), behavioural level (interaction protocols), quality of service level (non-functional properties

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

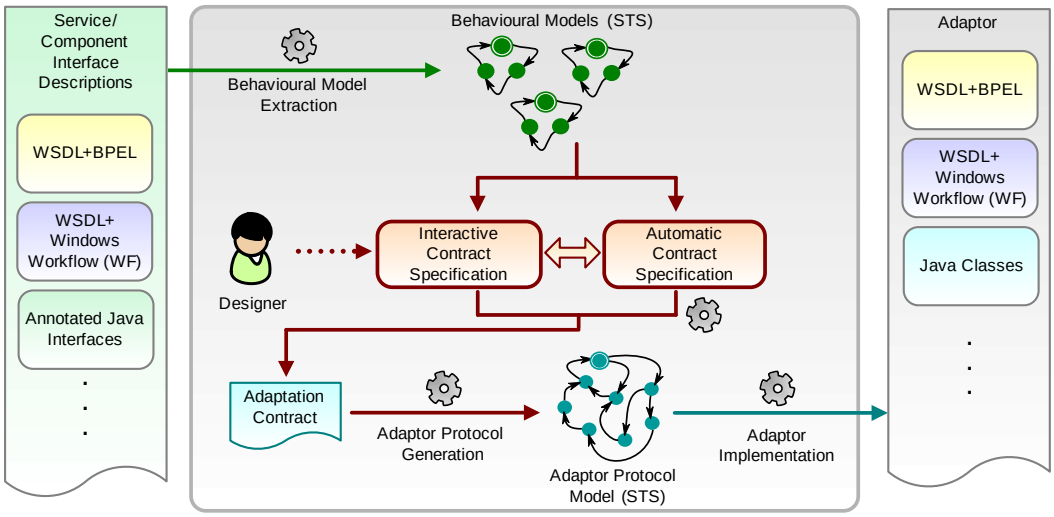

Fig. 1. Adaptation Process Overview

such as security or efficiency), and semantic or conceptual level (functional specification of what the service actually does). Recently, many academic approaches and industrial platforms have integrated behavioural descriptions in interface models and programming languages such as BPEL [1] or Windows Workflow Foundations [19] (.NET 3.0) in the context of Web services. Indeed, the behavioural interoperability level is essential [17], because even if services match from a signature point of view, their composition can lead to erroneous behaviours or deadlock situations if the designer is not aware of their execution flows, and does not take them into account while building a new system.

The kinds of mismatch cover in this work are: **(i)** mismatches in the name of the operations; **(ii)** *n* to *m* correspondences, where some messages in one interface must be matched against a different number of messages in another interface; and **(iii)** data mismatches, where there are incompatibilities in the number and/or type of arguments being sent or received. In order to solve these incompatibilities, adaptation contracts include a mapping between the operations (and their arguments) of the services to adapt.

The manual writing of an adaptation contract is a difficult and error-prone task. Incorrect correspondences between operations in service interfaces, or syntactic mistakes are common, especially when the contract has to be specified using cumbersome textual notations. Moreover, a contract is just an abstract specification of how the different services should interact and does not explicitly describe all the different execution scenarios of a system, which may not be easily envisioned by the designer. Finally, writing a contract requires a good knowledge of the services involved, and understanding all the details of service protocols is quite complicated for non-experts.

In this paper, we focus on the behavioural interoperability level, and present a semi-automatic approach to the specification of adaptation contracts. Specifically, we make use of an automatic contract generation engine, and an interactive contract specification environment to support the user through the adaptation contract design process. In order to achieve our goal, we propose a combined use of both elements. The automatic approach is able to generate deadlock-free contracts but it lacks the semantic information about the problem, therefore it suggests several contracts to the interactive environment, where the user is able to choose and customize the result. Our solution is completely tool-supported.

We applied our approach and tools to many examples for evaluation purposes, and we present our experimental results in this paper. Figure 1 gives an overview of the adaptation process and shows the stage where contract specification takes place. Let us note that the generation of adaptor protocols and code from the adaptation contract is out of scope here, and the reader interested in more details about that may refer to [9,15].

The rest of this paper is structured as follows: Section 2 presents our service model and contract specification language, as well as a case study which will be used to illustrate the different issues introduced throughout the remaining sections. Section 3 first describes the automatic contract generation algorithm that we use in our approach, and then presents how it is extended with an interactive environment for semi-automatic contract specification. Section 4 presents some experimental results that we used to assess the benefits of our approach. Finally, Section 5 reviews related work, and Section 6 draws up some conclusions.

2 Interface Model and Contract Specification Language

2.1 Interface Model

We assume that service interfaces are equipped both with a signature (set of required and provided operations) and a protocol represented by a *Symbolic Transition System* (STS) [1]. Communication between services is represented using *events* relative to the emission and reception of messages corresponding to operation calls. Events may come with a set of data terms whose types respect the operation signatures. In our model, a *label* is either the internal action τ or a tuple (M, D, PL) where M is the message name, D stands for the direction of communication (! for emissions and ? for receptions), and PL is either a list of data terms if the message corresponds to an emission, or a list of variables if the message is a reception.

Definition 2.1 [STS] A Symbolic Transition System or STS is a tuple (A, S, I, F, T) where: A is an alphabet which corresponds to the set of labels associated to transitions, 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 a transition relation.

This formal model has been chosen because it is simple, graphical, and it can be easily derived from existing implementation languages (see for instance [13,18,12] where such abstractions for Web services were used for verification, composition or adaptation purposes). For space reasons, in the rest of the paper, we will describe service interfaces only with their STSs. Signatures will be left implicit, yet they can be inferred from the typing of arguments (made explicit here) in STS labels.

Example. We describe a simple example which consists of a client and a supplier service. As it can be observed in Figure 2, the client first sends a request for an item to be purchased (`getItem!`), and receives its price (`getItem?`). Then, the client can either decide to `buy!` the item and receives a `confirmation?`, or `cancel!` the transaction. On the other side, the supplier waits for a product category (`setCategory?`) and a particular `itemRequest?`, and replies with the price of the requested item. After that, the transaction can either abort,

[1] In this paper, STSs are Labelled Transition Systems (LTSs) extended with value passing (data parameters coming with messages).

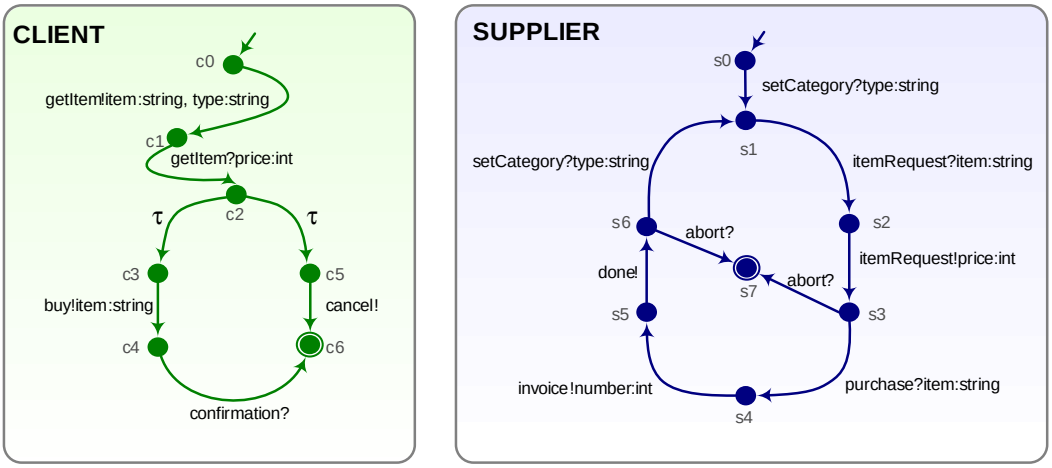

Fig. 2. Client and supplier behavioural interfaces

or receive the actual `purchase?` order, returning afterwards its corresponding `invoice!` and notifying the correct completion of the purchase (`done!`). Finally, execution can either finish (`abort?`), or continue with a new transaction.

Web service composition is subject to different mismatch situations: **(i)** name mismatch occurs if a service expects a particular message, and receives one with a different name (*e.g.*, the client sends `buy!`, whereas the supplier is expecting `purchase?`). **(ii)** n to m correspondence is given if a message on a particular interface corresponds to several ones in its counterpart's interface (or similarly, a message has no correspondence at all). In Figure 2 it can be observed that while the client intends to make an item request by only sending `getItem!`, the supplier on the other endpoint expects `setCategory?`, followed by `itemRequest?`. **(iii)** Data mismatch may occur when the number and/or type of arguments either being sent or received do not match between the events on the different interfaces. This can be observed in the supplier protocol when `invoice!` sends an invoice identifier but no argument is expected in its client counterpart (`confirmation?`).

2.2 Contract Specification Language

In this section, we present our adaptation language that makes communication among services explicit, and specifies how to work out mismatch situations. To make communication explicit, we rely on *vectors* (inspired from synchronization vectors [2]), which denote communication between several services, where each event appearing in one vector is executed by one service and the overall result corresponds to an interaction between all the involved services. A vector may involve any number of services and does not require interactions to occur on the same names of events. Vectors express correspondences between messages, like bindings between ports, or connectors in architectural descriptions. We consider a binary communication model, therefore vectors are always reduced to one event (when a service evolves independently) or two (when services communicate). Furthermore, variables are used as placeholders in message parameters. The same variable name appearing in different labels (possibly in different vectors) enables the relation of sent and received arguments of messages.

Definition 2.2 [Vector] A *vector* for a set of service $STS_i = (A_i, S_i, I_i, F_i, T_i), i \in \{1, .., n\}$

$$
\begin{aligned}
V = \{ v_{cat} &= \langle c\!:\!\mathsf{getItem!I,T};\ s\!:\!\mathsf{setCategory?T}\rangle,\\
v_{req} &= \langle s\!:\!\mathsf{itemRequest?I}\rangle,\\
v_{res} &= \langle c\!:\!\mathsf{getItem?P};\ s\!:\!\mathsf{itemRequest!P}\rangle,\\
v_{ab1} &= \langle c\!:\!\mathsf{cancel!};\ s\!:\!\mathsf{abort?}\rangle,\\
v_{ab2} &= \langle s\!:\!\mathsf{abort?}\rangle,\\
v_{done} &= \langle s\!:\!\mathsf{done!}\rangle,\\
v_{buy} &= \langle c\!:\!\mathsf{buy!I};\ s\!:\!\mathsf{purchase?I}\rangle,\\
v_{inv} &= \langle c\!:\!\mathsf{confirmation?};\ s\!:\!\mathsf{invoice!N}\rangle\}
\end{aligned}
$$

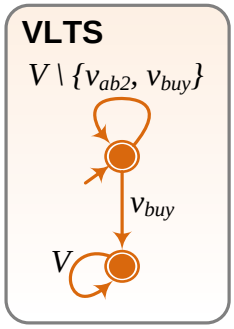

Fig. 3. Adaptation contract for our running example: vectors (left) and VLTS (right)

is an element of $A_j \cup (A_j \times A_k)$ with $j,k \in \{1,\ldots,n\}$, $j \neq k$. Such a vector is noted $\langle s_j\!:\!l\rangle$, or $\langle s_j\!:\!l, s_k\!:\!l'\rangle$ where s_j, s_k are service identifiers, and l, l' are labels on the alphabets of services A_j, A_k, where message parameters are substituted by placeholders relating the arguments.

In addition, the contract notation includes an LTS with vectors on transitions (vector LTS or VLTS). The purpose of VLTSs is to guide the application order of the interactions expressed by vectors. VLTSs go beyond port and parameter bindings, and express more advanced adaptation properties (such as imposing a sequence of vectors or a choice between some of them). If the application order of vectors does not matter, the vector LTS contains a single state and all transitions looping on it.

Definition 2.3 [Adaptation Contract] An adaptation contract for a set of services STS_i, $i \in \{1,..,n\}$, is a couple $(V, VLTS)$ where V is a set of vectors for services STS_i, and $VLTS$ is a vector LTS.

An adaptor protocol can be automatically generated from an adaptation contract using state-of-the-art techniques presented in [9,15]. Once the adaptor protocol is generated, it can be implemented using BPEL or Windows Workflow Foundation using techniques presented in [15,10].

Example. In order to illustrate how the different kinds of mismatch situations described in our example can be worked out, we focus on the initial part of the client and the supplier, where the item request is made: **(i)** name mismatch can be solved by writing the vector ⟨`c:getItem!I,T; s:setCategory?T`⟩; **(ii)** the correspondence established in the previous vector leaves `itemRequest?` on the supplier without counterpart. Thus, we can write another vector ⟨`s:itemRequest?I`⟩ in order to make the supplier service evolve independently; **(iii)** data correspondences are established through names `I, T, P` and `N`, which are used as placeholders by the designer in order to relate values (in emissions) with variables (in receptions).

Regarding the use of the vector LTS, in Figure 2 we can observe that at state c2, the client STS can either cancel the purchase or buy an item, whereas at s3, the supplier either waits for a purchase, or the cancellation of the operation (abort?). Up to the current state of the execution, vectors v_{cat}, v_{req}, and v_{res} have been fired. Now, vectors v_{ab1}, v_{ab2}, and v_{buy} could be fired. An evolution of the system through v_{ab1} would be correct, since the client and the supplier would reach their final states c6 and s7, respectively. In contrast, firing v_{ab2} at this stage would insert a deadlock in the system, since the client would be eventually blocked: the supplier would reach its final state s7, but the client would be stuck in c2 (no

more vectors available to be fired). On the right-hand side of Figure 3, we can observe how the execution of v_{ab2} is prevented by the VLTS in the contract until v_{buy} is executed, hence avoiding the potential deadlock of the system at state (c2,s3). Due to page limitation, we present the full description of the adaptor protocol generated for the contract in Figure 3 on Appendix A.

3 Semi-automatic Contract Specification

This section first presents our automatic generation process for adaptation contracts. Although this process is capable of generating correct contracts from the behavioural point of view, it cannot control the semantic constraints present on service interfaces. Hence, in the second part of this section we present a semi-automatic approach to contract specification as a solution to this problem. Rather than pursuing a fully automated approach by making use of semantic technologies (which may not be enough to infer complete interface operation semantics and their relations), our solution is based on extending our automatic generation process with an interactive contract specification environment that helps in customizing and constraining automatically generated solutions.

3.1 Automatic Contract Generation

Our engine for automatic contract specification (Algorithm 1) performs an incremental process where an initially empty contract is refined while traversing the service behaviours until a complete deadlock-free contract is generated. Our approach consists of a combination of an expert system and an informed-search algorithm.

Algorithm 1 *gen_contracts*

Returns a set of adaptation contracts for a couple of behavioural service interfaces.
inputs Service interfaces $STS_i = (A_i, S_i, I_i, F_i, T_i), i \in \{1, 2\}$ and an initial set of vectors V_0, empty by default.
output Set of adaptation contracts C

```
C, c, val = ∅, ε, 0
c0 = (V0, create_looping_VLTS(V0))
val0 = valuate_contract(c0)
queue = enqueue_contract(empty_queue(), val0, c0, I1, I2)
for val, c, s1, s2 = explore_contract(queue) and ¬is_complete(c) do
    {Generate all possible successor contracts for the given contract c.}
    for i = 1 to 2 do
        for all (si, a, s) ∈ T do
            c′ = refine_contract(c, a, i)
            val′ = valuate_contract(c′)
            if i = 1 then
                queue = enqueue_contract(queue, val′, c′, s, s2)
            else
                queue = enqueue_contract(queue, val′, c′, s1, s)
            end if
        end for
    end for
end for
if c ≠ ε then
    {Find in the queue other complete contracts with the same heuristic value.}
    C = {c} ∪ find_complete_contracts(queue, val)
end if
return C
```

Expert system. From a given partial contract (empty, by default) and a current state in every service behaviour (beginning with their initial states) the expert system generates new partial contracts by including every outgoing transition from those states in the given contract (*refine_contract*). These partial contracts compose a directed and acyclic graph whose initial node is the empty contract and, in every arc, the successor contract contains one label/message more than its predecessor (either included in a vector copied from the

parent contract or in a new vector with that single term). The expert system also contains rules (*is_complete*) that recognize which contracts are complete (*i.e.*, those which allow services to always reach a final state) and rules which evaluate every contract using a heuristic function (*valuate_contract*). Different scenarios and contexts require different adaptation policies, therefore the heuristic function and contract generation process are easily extended or customized by means of expert system rules. These rules can prune partial contracts in the graph depending on their vectors and the execution traces allowed by those contracts.

The heuristic function (see [14] for more details) is based on the direction of the operations and the matching between their arguments. It represents a measure of the suitability of the contract for service adaptation since it ranks first those contracts which synchronize compatible operations and avoid incompatible branches of the service behaviour. This function imposes a total order among the contracts and those which assume the minimum number of mismatches, *i.e.*, contracts where every operation corresponds to another and all the parameters match, are placed first.

Informed-search algorithm (A*). This algorithm accepts as input the graph of partial contracts generated by the expert system and it selects the next partial contract which has the lowest heuristic value (*explore_contract*). This selected partial contract is returned to the expert system to continue with the process until the $A*$ algorithm selects a complete contract (or several, if there are many with the same heuristic value, *find_complete_contracts*). In this way, the contract generation process is equivalent to a search, guided by the heuristic function, in the graph of partial contracts until a complete one is found. However, even though guiding the process with the heuristic function alleviates the state-explosion problem, the number of possible partial contracts increases exponentially with the size of the problem, therefore our automatic approach was originally designed to work with only two services at a time. $A*$ is an exhaustive search algorithm, therefore it always finds a solution. In the worst case, if the services are completely incompatible, a *trivial contract* will be generated where all emissions are ignored and all receptions are fulfilled with made-up arguments without any synchronization between the services.

3.2 Extending Automatic Generation with Interactive Specification

Service interfaces are not enough to automatically infer service functionality (*e.g.*, whether we are dealing with a booking or a weather service), nor the intended goal of their composition. When no additional requirements are given, deadlock-freedom is the only property preserved by the automatic approach. Therefore, contracts featuring undesirable behaviour (a service which always aborts the client's session, for instance) can be obtained in cases where such requirements are missing.

To solve this problem, we propose to extend automatic contract generation with an interactive specification environment that enables the customization of automatically generated solutions, incorporating the following elements:

Graphical notation. Based on the model described in Section 2, this notation is used in order to: **(i)** Visualize service interfaces. As it can be observed in Figure 4, the graphical notation for a service interface includes a representation of its protocol (STS) and a collection of ports. Each label on the STS corresponds to a *port* in the graphical description. Ports include a *data port* for each parameter contained in the parameter list of the label. **(ii)** Define port bindings. Correspondences between the different service interfaces (vec-

tors) are represented as *port* and *data port bindings* (solid and dashed lines, respectively). Starting from the graphical representation of the interfaces, the designer builds a contract by successively connecting ports and data ports. This results in the creation of bindings which specify how the interactions should be carried out. It is also possible to add a *port cap* (vector with a single label) on a port in order to indicate that it does not have to be connected anywhere. Port caps are represented by a "⊣" on the corresponding port. Moreover, our graphical notation supports the incremental specification of the adaptation through the encapsulation of service hierarchies inside *composite services*. This is particularly useful in cases where the number of services involved in the composition is high.

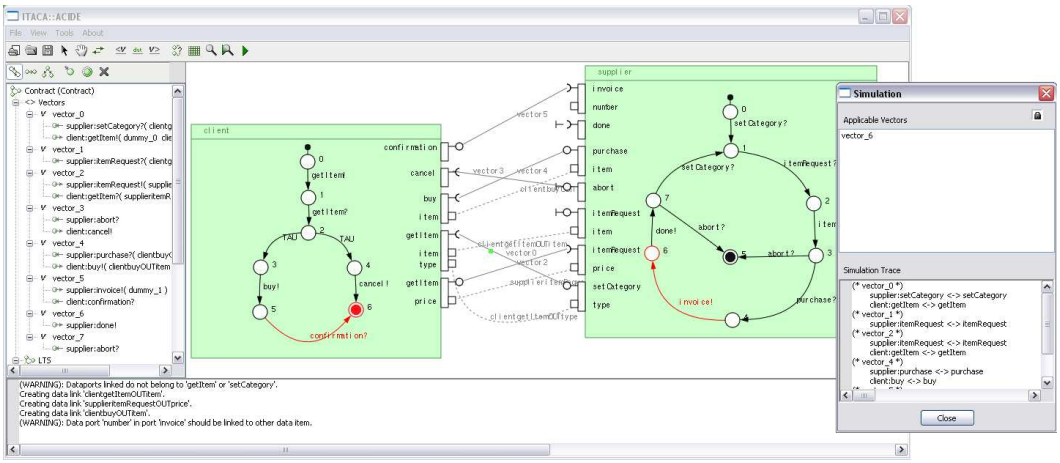

Fig. 4. Interactive contract specification and simulation for our case study

Verification and validation techniques. In order to help the designer to understand if the behaviour of the system complies with his/her design intentions, we provide fully automated techniques: **(i)** Simulation. Our environment implements an algorithm able to determine how the different behavioural interfaces evolve step-by-step as different vectors in the contract are executed; and **(ii)** Trace-checking. Potential system execution traces are first generated, and then traversed to detect those leading to deadlock situations or infinite loops.

This interactive environment can provide additional constraints (expressed as vectors) as input to the automatic approach which must be respected during the automatic generation process. Furthermore, the designer can take advantage of automatically generated vectors using the interactive environment in two ways: **(i)** taking them as suggestions when designing the contract from scratch. This is particularly interesting in scenarios that present behavioural interfaces with large protocols and only a few incompatibilities, where the designer must connect all ports one by one even if they obviously match with each other; or as **(ii)** complementing an already existing partial specification, enabling incremental contract specification.

Specifically, we propose an iterative process for contract refinement that integrates our automatic generation process (Algorithm 2) into the interactive contract design environment. Initially, we may optionally impose some vectors as constraints (Line 6) which may be directly related with the intended goal. Then, we invoke the automatic generation process (*gen_contracts* in Line 8) with the given initial vectors as input and we select through

Algorithm 2 *contract_specification*

Builds an adaptation contract for a set of behavioural service interfaces.
inputs Service interfaces $STS_i = (A_i, S_i, I_i, F_i, T_i), i \in \{1, \ldots, n\}$
output Adaptation contract $c = (V, VLTS = (V, S_{VLTS}, I_{VLTS}, F_{VLTS}, T_{VLTS}))$

1: $s = new_state()$
2: $V = \emptyset$
3: $VLTS = (V, \{s\}, s, \{s\}, \emptyset)$
4: $c = (V, VLTS)$
5: **while** $\neg env_valid(c)$ **do**
6: $\quad V_{rest} = env_input_vectors()$
7: $\quad (STS_a, STS_b) = env_select_STSs(STS_1, \ldots, STS_n)$
8: $\quad V_{gen} = env_select(gen_contracts(STS_a, STS_b, V \cup V_{rest}))$
9: $\quad$ **for all** $v_g \in V_{gen}$ **do**
10: $\quad\quad$ **if** $\neg env_valid(v_g)$ **then**
11: $\quad\quad\quad V_{gen} = V_{gen} \setminus \{v_g\}$
12: $\quad\quad$ **end if**
13: $\quad$ **end for**
14: $\quad V_{add} = env_input_vectors()$
15: $\quad V_{\Delta} = V_{add} \cup V_{gen}$
16: $\quad$ **for all** $v_{\delta} \in V_{\Delta}$ **do**
17: $\quad\quad V = V \cup \{v_{\delta}\}$
18: $\quad\quad T_{VLTS} = T_{VLTS} \cup \{s, v_{\delta}, s\}$
19: $\quad$ **end for**
20: $\quad (VLTS, s) = env_edit_VLTS(VLTS)$
21: $\quad V = env_edit_vectors(V)$
22: $\quad c = (V, VLTS)$
23: **end while**
24: **return** c

$$
\begin{aligned}
V = \{v_{cat} &= \langle c\!:\!\mathsf{getItem!I,T};\ s\!:\!\mathsf{setCategory?T}\rangle,\\
v_{req} &= \langle s\!:\!\mathsf{itemRequest?I}\rangle,\\
v_{res} &= \langle c\!:\!\mathsf{getItem?P};\ s\!:\!\mathsf{itemRequest!P}\rangle,\\
v_{ab1} &= \langle c\!:\!\mathsf{cancel!};\ s\!:\!\mathsf{abort?}\rangle,\\
v_{bad} &= \langle c\!:\!\mathsf{buy!};\ s\!:\!\mathsf{abort?}\rangle,\\
v_{conf} &= \langle c\!:\!\mathsf{confirmation?}\rangle\}
\end{aligned}
$$

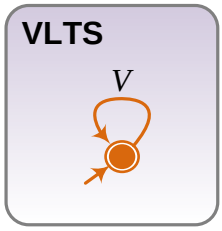

Fig. 5. Incorrect deadlock-free contract for our case study

the interactive environment one of the returned contracts. After validating the resulting contract using the verification mechanisms provided by the interactive environment, we determine if the current state of the contract is satisfactory (Line 5). If that is the case, the specification process ends. Otherwise, we may alternatively: (i) remove the parts of the contract causing problems (Lines 9–13); or (ii) customize the parts of the contract causing problems using the interactive environment (Lines 14–21).

We now informally define the functions used by our algorithm. It is worth noticing that all functions with names starting by *env* correspond to functionality implemented in the interactive environment which requires user input. Function *new_state* creates a new state identifier for the VLTS. *env_valid* returns whether the user considers the given vector valid or not. *env_input_vectors* returns a set of vectors composed by the user. *env_select_STSs* returns a pair of STSs. *env_select* returns a set of vectors selected from a set of contracts. *gen_contracts* (detailed in Algorithm 1) receives a pair of service STSs and a partial contract that will be used as restrictions (may be empty), and returns different contracts connecting the interfaces. *env_edit_VLTS* receives the current VLTS, and returns an edited version, and the new active VLTS state selected by the user. *env_edit_vectors* receives a set of vectors, and returns its edited version.

Example. The automatic approach generates two contracts with the same heuristic value for our running example. The first contract (Figure 3) is the one defined as an example in Subsection 2.2. However, the second contract (Figure 5) is not desirable because v_{bad} matches the client's `buy!` request with the `abort?` branch in the supplier, *i.e.*, the supplier

will abort the session no matter what the client decides. This second contract avoids the argument mismatch occurring in the first contract with v_{inv} (to receive and discard the invoice number received from the supplier) because the heuristic function considers this mismatch as bad as ignoring the argument received from the client's buy! in v_{bad}, therefore it aborts the purchase. However, if we impose vector v_{buy} in Figure 3 as a compositional constraint, the automatic approach restricts the generation process to contracts containing that vector and is able to fulfill the rest of the contract correctly.

4 Tool Support and Experimental Results

Our approach has been fully implemented and included into ITACA [8] (Integrated Toolbox for the Automatic Composition and Adaptation of Web Services). ITACA is a toolbox implemented at the University of Málaga that fully covers the adaptation process which goes from behavioural model extraction from existing service interface descriptions, to the final adaptor implementation.

In order to assess the benefits of our approach to contract specification in terms of development effort and contract accuracy, we conducted an experimental study with the help of a group of volunteers who were divided in three categories (expert, average, novice) according to their expertise and familiarity with behavioural interfaces and software composition. The tests consisted in handing over to the volunteers adaptation problems which included the graphical description of the behavioural interfaces to be reused in the composition and a short specification in natural language of what was the intended functionality of the system. Since we measure user productivity in our experiments, the automatic approach as an independent tool is left out of our study. The three different approaches for contract specification included in the experiments were: **(i)** manual contract specification (M), where the user had to write down the contract without further assistance; **(ii)** interactive contract specification (I), where volunteers made use of our interactive environment; and **(iii)** Semi-automatic contract specification (A+I), where the user specified the contract using the approach presented in this paper.

					Time (s)			Errors		
Problem	**Interf.**	**Ports**	**States**	**Trans.**	**M**	**I**	**A+I**	**M**	**I**	**A+I**
ftp-002	2	9	11	11	338	222	130	1.77	1.5	0
client-sup-002	2	12	15	16	480	248	183	0.33	0.5	0
which-004	2	17	16	19	486	146	126	2.95	0.75	0
online-med-003	3	15	16	17	531	189	122	5	0	2
easyrest-005	4	17	22	24	689	310	203	3	1.66	1.5
pda-001	6	46	37	48	2160	1152	1087	27.6	10.66	13.33

Table 1
Problem size and experimental results for the three approaches.

For our experiments we used different case studies that were obtained from our own archive of adaptation problems, which includes examples ranging from small synthetic ones to real-world case studies. Table 1 summarizes the problems used for our study, which are organized according to increasing size and complexity. We also include the number of services involved and ports to connect, as well as the overall size of the protocols (total number of states and transitions). The table also shows the experimental results (time required to solve the problem and number of errors in the specified contract) for each of the case studies and tested approaches.

Time spent. Figure 6 shows the results of our experiments. If we take a look to the left-hand side of the figure, we can observe that there is a remarkable difference in the amount of time required to solve the different problems between manual specification and the interactive approach (an average improvement of 53% using interactive specification). In addition, it is worth considering that users spent a reasonable amount of time simulating and validating contracts with the interactive environment whereas they did not when manually designing the same contracts.

Comparing interactive specification and the semi-automatic approach, there is an additional reduction in the amount of time required when the semi-automatic approach is used (12% on average). However, as the number of services to compose in the problem increases, this difference between the semi-automatic and interactive approaches is noticeably reduced (from 27.2% in the simplest case study ftp-002, to 3% in the most complex one pda-001). This is due to the fact that the automatic approach is only able to consider two interfaces at a time and, as the number of interfaces increases, the user has to select more pairs of interfaces to generate bindings between them, adding an additional complexity to the task.

Effort and accuracy. Regarding the accuracy during the adaptation process, we measured as errors the number of bindings created between ports which were either wrong or useless for the resulting contract. In addition, we also considered the number of mistakes remaining in the resulting contracts. In the case of manual specification we also took into account syntactic errors (tool-supported approaches avoid this kind of mistakes). In Figure 6 (right), it can be noticed that the number of errors in problem solutions tends to be smaller in tool-supported approaches compared to manual contract specification (an average improvement of 59% and 77% over manual specification with interactive and semi-automatic specification, respectively). This improvement increases with the complexity of the problem.

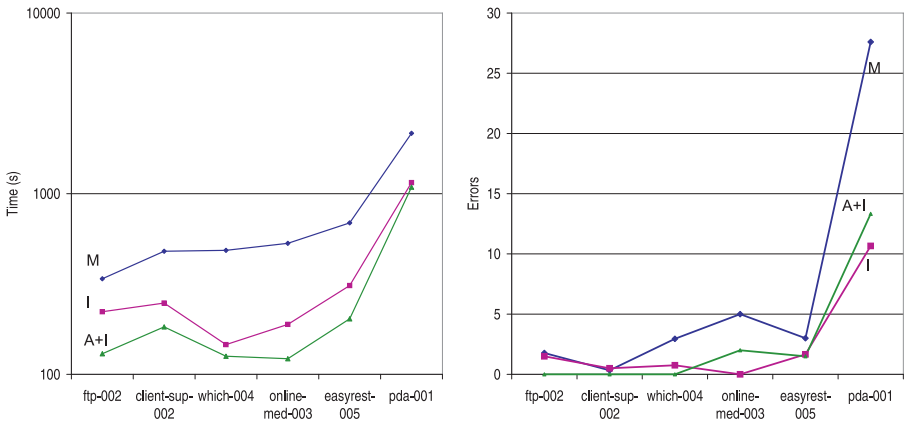

Fig. 6. Experimental results: Time elapsed (left) and accuracy (right)

If we focus on the comparison between the tool-supported approaches, the semi-automatic approach minimizes the number of errors in problems which contain only two interfaces. This happens because the automatic approach generates the majority of the bindings required to solve the problem, and the user does only have to customize the so-

lution if it is required, focusing on the remaining details. As it happened in the case of time, in problems where multiple small services have to be adapted, this improvement is lost since the user must modify or create additional bindings to integrate all the pairwise bindings returned by the automatic approach.

It is worth observing that in the case of online-med-003, the general trend between the interactive and the semi-automatic approach is reversed since users always solved the problem correctly in the first attempt using the interactive approach but, in the case of the semi-automatic approach, they need to modify two bindings on the contract returned by the automatic approach to integrate the third service of the example. In easyrest-005, the semi-automatic approach hardly improves (5.3%) the result of interactive specification since the problem contains two main interfaces which can be related using the automatic approach, leaving details to the user.

5 Related Work

Model-based behavioural adaptation approaches are often classified in two families: (i) automatic approaches that are fully automated and try to solve interoperability issues by pruning the behaviours that may lead to mismatch, and (ii) generative approaches that are able to accommodate protocols, for instance by reordering messages and their parameters, or by supporting the specification of advanced adaptation scenarios. In this section, we compare our solution with existing automatic approaches, and generative ones especially those supporting the designer in the contract specification.

Automatic contract specification. The authors of [7] outlined a methodology for the automatic generation of adaptors capable of solving behavioural mismatches between BPEL processes. In their adaptation methodology they use YAWL as an intermediate workflow language. Once the adaptor workflow is generated, they use lock analysis techniques to check if a full adaptor has been generated or only a partial one (some interaction scenarios cannot be resolved). They solve protocol incompatibilities but their approach does not address signature mismatch since they assume same operation names (and arguments) among the services. In [4], the authors address the enforcement of certain behavioural properties (namely liveness and safety properties expressed as LTL properties) out of a set of already implemented components. Starting from the specification with MSCs of the components to be assembled and of the properties that the resulting system should verify, they automatically derive deadlock-free adaptor glue code for the set of components in order to obtain a property-satisfying system. However, although this approach enables a precise specification of the desirable behaviour of the system, it works by pruning branches of the behaviour which are incompatible or do not satisfy the specified properties. Hence, the range of situations where mismatch can be reconciled is limited compared to other approaches.

Interactive contract specification. Brogi *et al.* [6] present a methodology for generative behavioural adaptation where component behaviours are specified with a subset of the π-calculus and composition specifications with name correspondences. An adaptor generation algorithm is used to refine the given specification into a concrete adaptor which is able to accommodate both message name and protocol mismatch. More recently, [9,15] proposed state-of-the-art adaptation approaches that are generative and support adaptation policies and system properties described by means of regular expressions or LTSs of vectors. However, in these works, no support is proposed to help the designer during the

contract specification task, which is therefore achieved manually. Dumas *et al.* [11] introduce an approach to service interface adaptation using a visual language based on an algebra over behavioural interfaces. A graphical editor taking as input pairs of behavioural interfaces allows to link them through interface transformation expressions. The output of this tool can be used as input for a service mediation engine which interprets the information in order to perform composition. Although this approach provides the means to define interface transformation expressions graphically, it does not support the incremental specification of adaptation since it only considers pairs of provided-required interfaces. Moreover, our approach provides systematic contract verification mechanisms.

Automatic-interactive approach. To the best of our knowledge, [16] is the only work mixing both automatic and interactive aspects while building adaptation contracts. In [16], some techniques are presented to automatically match the WSDL signature of two Web services. The matching is performed by a combination of an XML schema matching tool called COMA++ [3] and some protocol analysis. They are able to generate a mismatch tree that gathers all protocol mismatches, and ask the designer to give a mapping function in order to solve these mismatches if they are not automatically adaptable. However, no support is provided to help the designer to specify this mapping function whereas we propose a full environment to guide him in this task.

6 Conclusions

Manual specification of adaptation contracts is a cumbersome and error-prone task. In this paper, we proposed a novel solution to ease the task of contract specification. The proposed approach is semi-automatic, and relies on an interactive environment and automatic generation techniques to support the designer. Our solution has been fully implemented in tools, which have been applied to many case studies. Furthermore, we have shown that our approach remarkably reduces the time spent to build the contract, as well as the number of errors made during the process. More concretely:

- The time required to specify adaptation contracts using our approach has been reduced to 35% of the overall time required to manually specify the contract.
- Our approach yields an accuracy improvement of 77% relative to manual contract specification.
- Our proposal worked especially well in cases where functionality is not scattered across multiple small interfaces.

As regards future work, we aim at extending our approach to consider goal-oriented adaptation, using as input to the adaptation process a high-level property written using temporal logic, that will be used to guide the contract construction.

Acknowledgements. This work has been partially supported by the project TIN2008-05932 funded by the Spanish Ministry of Innovation and Science (MICINN).

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] Arnold, A., "Finite Transition Systems," International Series in Computer Science, Prentice-Hall, 1994.

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

[4] 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.

[5] Becker, S., A. Brogi, I. Gorton, S. Overhage, A. Romanovsky and M. Tivoli, *Towards an Engineering Approach to Component Adaptation*, in: *Architecting Systems with Trustworthy Components*, LNCS **3938** (2006), pp. 193–215.

[6] Bracciali, A., A. Brogi and C. Canal, *A Formal Approach to Component Adaptation*, Journal of Systems and Software **74** (2005), pp. 45–54.

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

[8] Cámara, J., J. A. Martín, G. Salaün, J. Cubo, M. Ouederni, C. Canal and E. Pimentel, *ITACA: An Integrated Toolbox for the Automatic Composition and Adaptation of Web Services*, Proc. of ICSE'09 (2009), pp. 627–630.

[9] Canal, C., P. Poizat and G. Salaün, *Model-Based Adaptation of Behavioural Mismatching Components*, IEEE Transactions on Software Engineering **34** (2008), pp. 546–563.

[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 **215** (2007), pp. 39–55.

[11] Dumas, M., M. Spork and K. Wang, *Adapt or Perish: Algebra and Visual Notation for Service Interface Adaptation*, in: *In Proc. of BPM'06*, LNCS **4102** (2006), pp. 65–80.

[12] Foster, H., S. Uchitel and J. Kramer, *LTSA-WS: A Tool for Model-based Verification of Web Service Compositions and Choreography*, in: *Proc. of ICSE'06* (2006), pp. 771–774.

[13] Fu, X., T. Bultan and J. Su, *Analysis of Interacting BPEL Web Services*, in: *Proc. of WWW'04* (2004), pp. 621–630.

[14] Martín, J. A. and E. Pimentel, *Automatic Generation of Adaptation Contracts*, in: *Proc. of FOCLASA'08*, ENTCS **229** (2009), pp. 115–131.

[15] Mateescu, R., P. Poizat and G. Salaün, *Adaptation of Service Protocols using Process Algebra and On-the-Fly Reduction Techniques*, in: *Proc. of ICSOC'08*, LNCS **5364** (2008), pp. 84–99.

[16] 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.

[17] Plasil, F. and S. Visnovsky, *Behavior Protocols for Software Components*, IEEE Transactions on Software Engineering **28** (2002), pp. 1056–1076.

[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] Scribner, K., "Microsoft Windows Workflow Foundation: Step by Step," Microsoft Press, 2007.

[20] Yellin, D. M. and R. E. Strom, *Protocol Specifications and Components Adaptors*, ACM Transactions on Programming Languages and Systems **19** (1997), pp. 292–333.

A Adaptor Protocol

Figure A.1 displays the adaptor protocol generated using the adaptation contract described in Subsection 2.2. For illustration purposes, our example is rather simple and in this case, the adaptor protocol contains only 18 states and 19 transitions (although they tend to be typically quite large). Interaction starts by receiving the category and the item to purchase from the client. Next, the adaptor (state 5) can alternatively: (i) receive `buy` and perform the purchase; or (ii) receive `cancel` from the client and issue `abort` to the supplier before finishing (state 11). It is worth observing that abort cannot be executed without the client's cancellation at this point, and it can only occur on its own after the purchase is made, according to the constraints expressed in the VLTS (Figure 3). The part of the adaptor after state 10 corresponds to the confirmation of the purchase and the end of the transaction.

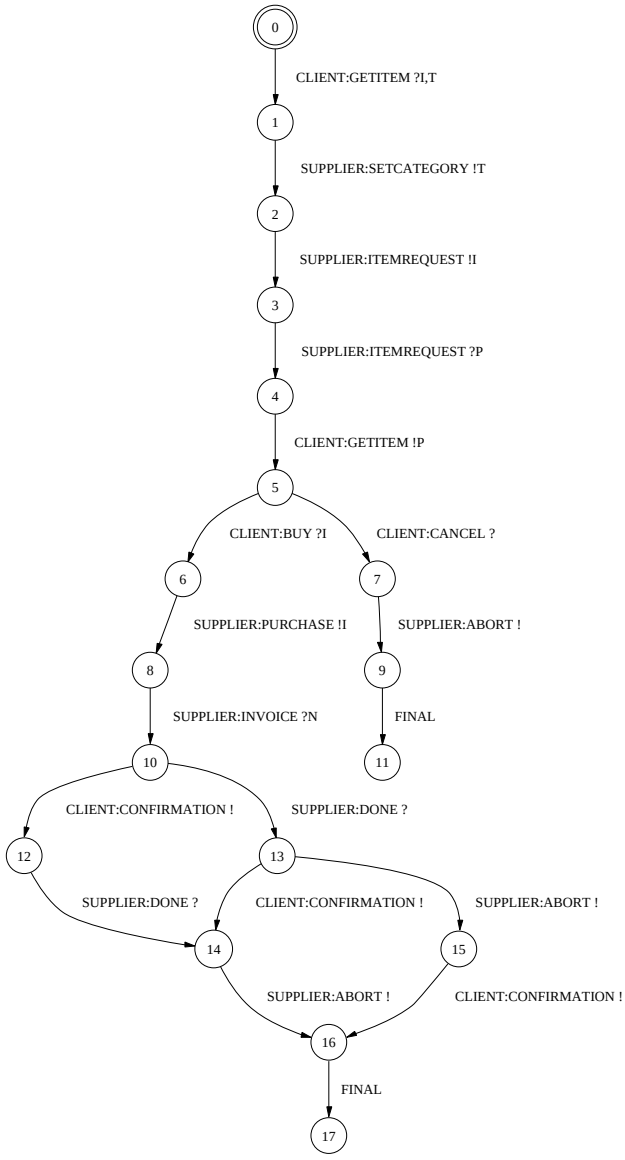

Fig. A.1. Adaptor protocol generated for our running example