An Automatic Approach to Enable Replacement of Conversational Services*

Luca Cavallaro[1], Elisabetta Di Nitto[1], and Matteo Pradella[2]

[1] Politecnico di Milano, DEI, Piazza L. Da Vinci, 32, 20133 Milano, Italy
{cavallaro,dinitto}@elet.polimi.it
[2] CNR IEIIT-MI, Via Golgi, 40, 20133 Milano, Italy
pradella@elet.polimi.it

Abstract. In Service Oriented Architectures (SOAs) services invoked in a composition can be replaced by other services, which are possibly discovered and bound at runtime. Most of the research efforts supporting this replacement assume that the interface of the interchangeable services are the same and known at design time. Such assumption is not realistic since it implies that providers of the same kinds of services agree on the interfaces the services offer. By *interface mapping* we mean the class of approaches aiming at relaxing this assumption. Most of those approaches available in the literature focus on stateless services and simply address mapping operation names and data structures. Instead, this paper focuses on *conversational services* for which the sequence of required operation calls, i.e., the *interaction protocol*, matters. We use model checking to automatically identify the interaction protocols mapping. We validate our technique both by applying it to the invocation of two real services (Flickr and Picasa), and by quantitatively comparing it to a related approach.

1 Introduction

Service oriented architectures (SOAs) offer the mechanisms to build software systems integrating loosely coupled services, possibly made available by third party vendors. As services may be controlled by third parties, they may be out of service consumers control. This means that the traditional closed world assumption, which mandates that developers know a priori all the components involved in the system and can model and plan their interactions, is no more verified [1] because services involved in the composition may change during the system life cycle to react to failures and service unavailabilities. When this happens, a new service semantically equivalent to the one not responding properly could be discovered and bound to the composition. When this replacement occurs at runtime, the composition (or the framework where the composition is running) should be able to perform the replacement requiring as little human intervention as possible.

In recent years, research about service oriented architectures produced many frameworks that can provide run time reconfigurations of service compositions (see for instance [2], [3]), but most of them make the hypothesis that all semantically equivalent

* This research has been funded by the European Community's FP7/2007-2013 Programme, grant agreement 215483 (S-Cube), and IDEAS-ERC Programme, Project 227977 (SMSCom).

L. Baresi, C.-H. Chi, and J. Suzuki (Eds.): ICSOC-ServiceWave 2009, LNCS 5900, pp. 159–174, 2009.

services have the same interface. This hypothesis, however, is not realistic as services can be released by independent vendors. Therefore in common practice interfaces lack standardization. Consequently there is no guarantee that services discovered and bound at runtime can perfectly fit in a preexistent composition.

To address this problem, in a previous work [4], we have developed an approach to allow invocation of services whose interfaces and behaviors differ from each other. The approach was based on the definition of proper *mapping scripts* that, when interpreted at runtime, could solve complex mismatches and perform the needed adaptations.

In this paper we extend the previous work by providing an approach and a tool to support the automation of the mapping scripts definition. The approach is able to handle *conversational services*, that is, services whose operations are expected to be called in some specific sequences, which define the services *interaction protocols*. We assume that, when developing a service composition, a service integrator uses the component services that are available at the time he/she is developing the system. We call these *abstract services* to highlight the fact that they are not necessarily the ones that will be actually used at runtime, which we call *concrete services*. We also assume that services are described not only in terms of their syntactic interface (i.e., their WSDL or any equivalent description), but also in terms of a model that defines the order in which service operations need to be invoked.

Given these assumptions and given a certain sequence of operations to be invoked on an abstract service, our approach is able to propose a possible mapping of this sequence to a sequence of operations on a concrete service. The result of this analysis is a mapping script fragment that, combined with other fragments that deal with data and operation names mappings, allows us to actually adapt abstract service invocations to their concrete implementations. Data and operation names mappings are disregarded in this paper as they appear to be much simpler than the mapping of interaction protocols and are covered in [5] and in other approaches in the literature (see Section 2).

The rest of the paper is organized as follows: Section 2 presents the current state of the art and highlights some open issues. Section 3 presents a real world example that motivates our work, Section 4 summarizes the background work that has been developed in [4], Section 5 discusses our approach to support semi-automatic generation of mapping scripts for what concerns protocol-level mismatches and refines the execution model associated to this specific case of mismatches. Section 6 evaluates our approach quantitatively and qualitatively, and, finally, Section 7 draws some conclusions.

2 Related Work

The approaches that support interface mapping can be categorized in those that require human intervention in the definition of mapping scripts or equivalent mechanisms (see for instance [6], [7], [8] and [9]) and those that offer some automatic tool.

Among the approaches in the first category, we mention here the one in [7] as it offers a model checking approach to verify the correctness of *adaptation contracts* that are manually defined by humans, and the one in [9] as it assists humans in the interface adapters development by offering a tool that provides hints about possible mismatches between an abstract and a concrete service interface. Both approaches, however, assume

that, before execution, a developer can identify all potential pairs of abstract and concrete services and specify all needed adapters. This may not work properly in the cases of systems supporting run-time substitutions of services as the substitutions could have not be foreseen in advance.

Automated approaches try to solve this issue by generating adapters that are inferred from specifications associated to services. Many of these approaches are based on the use of ontologies. Among the others, our previous work [5] and [10] exploit a domain ontology (specified in SAWSDL[1]) to annotate service interfaces. At run-time, when a service bound to a composition needs to be substituted, a software agent generates a mapping by parsing the ontological annotations in the interfaces. *SCIROCO* [11] offers similar features focusing on stateful services. It requires all services to be annotated with both a SAWSDL description and a WS-ResourceProperties[2] document, which represents the state of the service. When an invoked service becomes unavailable, *SCIROCO* exploits the SAWSDL annotations to find a set of candidates that expose a semantically matching interface. Then, the WS-ResourceProperties document associated to each candidate service is analyzed to find out if it is possible to bring the candidate in a state that is compatible with the state of the unavailable service. If this is possible, then this service is selected for replacement of the one that is unavailable. All of these three approaches offer full run-time automation for service substitution, but can address only those mismatches that concern data and operation names while they disregard those concerning the interaction protocol.

An approach that generates adapters covering the case of interaction protocols mismatches is presented in [12]. It assumes to start from a service composition and a service behavioral description both written in the BPEL language [13]. These are then translated in the *YAWL* formal language [14] and matched in order to identify an invocation trace in the service behavioral description that matches the one expected by the service composition. The matching algorithm is based on graph exploration and considers both control flow and data flow requirements.

The approach presented in [15] offers similar features and has been implemented in an open source tool.[3] While both these approaches appear to fulfill our need for supporting interaction protocol mapping, they may present some shortcoming in terms of performances due to the high cost of exhaustive graph exploration algorithms that could prevent their usage in on-the-fly mapping derivation. While no data about performances are available for the approach in [12], we could exploit the tool offered by [15] to verify our guess. As discussed in Section 6, the processing time required by the tool is remarkably high in complex cases. Our goal is, therefore, to exploit some alternative technique to significantly improve these performances.

3 Motivating Example

To motivate our work we refer to an example based on some significant conversational services available on the Internet. Our example application is a photo management tool

[1] http://www.w3.org/2002/ws/sawsdl/

[2] http://docs.oasis-open.org/wsrf/wsrf-ws_resource_properties-1.2-spec-os.pdf

[3] http://sourceforge.net/projects/dinapter

designed for working on a mobile phone. A user can take some photos with his mobile, upload them to the web, and share them with his friends using an external service.

The tool expects to interact with the *Flickr* service[4]. *Flickr* makes available to its users a space where to upload photos and a REST[16] service to access it. Photos can have assigned one of the following levels of visibility: public, private, and family, where the latter lets only some members see the photos uploaded by a user. Once the user has uploaded some photos the service lets him group (part of) them in sets. Of course it is always possible to change the visibility of a photo or of a set.

Flickr is not the only service offering a photo repository. Another analogous service is called *Picasa*[5]. *Flickr* and *Picasa* are equivalent in a broad sense, but analyzing their interfaces in more detail some differences emerge. In particular, *Picasa* does not support the upload of photos if they are not grouped in a set. For this reason a user should first create a set and then upload pictures directly into the created set. In addition, while *Flickr* identifies three levels of visibility for photos and sets, *Picasa* only supports two (private and public) and, given the central role of sets, associates these levels only to sets and not directly to photos. Of course, other differences concern the names and the parameters of the equivalent operations made available by the two services. For instance the operation *addToSet* of *Flickr* and the operation *createPhoto* of *Picasa* both add a photo to a set, but they show different names and accept different input parameters (Tables 1 and 2 summarize the Flickr and Picasa operations we focus on.

Even if our photo management tool is built to be used with *Flickr*, many users may be subscribed to *Picasa* or to any other popular photo sharing service. In order to allow them to use any of these alternative services, either we hardcode in our tool the instructions to interact with any possible service, or we build a mapping mechanism that handles the mismatches on our behalf. Such mapping mechanism could state, for example, that the sequence of *Flickr* operations *uploadPriv*, *addToSet*, *makeSetPub* maps on the following sequence of *Picasa* operations: *createPublicSet* and *createPhoto*, which can therefore be invoked to obtain the required behaviour. The approach we discuss in this paper is focusing specifically on how to automatically and efficiently infer such kinds of mapping without or with limited human intervention.

4 Adaptation Approach: Overview

In order to describe possible differences that can arise between an abstract and a concrete service we need to define our model of a service. A service can be described as a *Labeled Transition System* (LTS) characterized by tuple $P = (S, O, \tau)$, where:

- S is the set of states the service can go through.
- O is the set of operations that can be invoked on the service together with the corresponding parameters. In formal terms, this is the input alphabet of the LTS.
- τ is the transition function $\tau : S \times O \rightarrow 2^S$ that describes how the service can evolve from state to state when operations are invoked. 2^s indicates that the transition function can non-deterministically lead the service to different states depending on the context (e.g., a state representing a correct functioning of the service

[4] http://www.flickr.com/services/api/

[5] http://code.google.com/apis/picasaweb/overview.html

Table 1. A subset of *Flickr* operations and required data

Operation name	*Parameters*	*Return value*	*Description*
uploadPub	photo photoName	success	Uploads a photo with *public* visibility
uploadPriv	photo photoName	success	Uploads a photo with *private* visibility
uploadFam	photo photoName	success	Uploads a photo with *family* visibility
makePhotoPub	photoName		Makes a photo visibility *public*
makePhotoPriv	photoName		Makes a photo visibility *private*
makePhotoFam	photoName		Makes a photo visibility *family*
addToSet	albumName photoName	success	Adds a previously uploaded photo to a (new or existent) set
makeSetPub	albumName		Makes a set visibility *public*
makeSetPriv	albumName		Makes a set visibility *private*
makeSetFam	albumName		Makes a set visibility *family*

Table 2. A subset of *Picasa* operations and required data

Operation name	*Parameters*	*Return value*	*Description*
createPublicSet	albumName	success	Creates a photo set with *public* visibility
createPrivateSet	albumName	success	Creates a photo set with *private* visibility
createPhoto	albumName photoName photo	success	Uploads a new photo and adds it to an existent set
makePub	albumName		Makes a set visibility *public*
makePriv	albumName		Makes a set visibility *private*

can be reached only after the user has been identified, otherwise an error state has to be reached), or on possible service failures (e.g., when an a timeout expires the corresponding transition leads to an error state).

Each operation $o \in O$ is a triple $\langle name, in, out \rangle$, where $name$ is the operation name, in and out are possibly empty multisets of data the operation requires as input and returns as output, respectively. A datum is a triple $\langle name, type, value \rangle$. $name$ is the name of the datum, $type$ is the type of the datum and $value$ is the value that the datum assumes.

Given an abstract and a concrete service, we say that a *mismatch* occurs when an operation request expressed in terms of the abstract interface cannot be understood by the concrete service that should handle it. We distinguish between two mapping classes:

- *Interface-level mismatches* concern differences between names of operations exposed by an abstract and a concrete service and parameters of these operations.
- *Protocol-level mismatches* concern differences in the order the operations offered by an abstract service and by its concrete representation are expected to be invoked.

As discussed in Section 2, interface-level mismatches have been threated in the literature and addressed either through methodological approaches involving human designers [9] or through automatic approaches able to reason in the presence of some reference ontology [10,11]. Thus, we do not go into further details on this aspect and handle it by exploiting the approach we reported in [5].

Protocol-level mismatches are those we want to focus on in this paper. As mentioned before, they apply to stateful conversational services for which the sequence in which operations are invoked matters. In this case, we can distinguish between the following classes of mismatches:

- *One to one binding*: an operation in the abstract service has a direct counterpart in the concrete service that can replace it. This case is addressed directly as an interface-level mismatch and therefore is not further considered in this paper.
- *One to many binding*: an operation in the abstract service does not have a direct counterpart in the concrete service but it can be mapped into two or more of its operations.
- *Many to one binding*: two or more operations in the abstract service do not have a direct counterpart in the concrete services, but, all together, can be mapped into one operation of the concrete service.
- *Many to many binding*: a sequence of operations on the abstract service can be mapped into a different sequence of operations on the concrete service.

Our aim is to focus on the general case of many to many binding and, based on it, deal also with the simpler cases. In particular, we aim at defining *mapping scripts* that contain histories which associate sequences of operations on the abstract services into sequences of operations on the corresponding concrete services.

At runtime, the mapping scripts are interpreted by adapters that are then able to invoke concrete services thus overcoming their mismatches with respect to the abstract services. Figure 1 shows the main components of our runtime infrastructure. Also, it shows how these components interact when a service composition tries to call a sequence of operations of an abstract service S1 and this sequence is then translated into a sequence of operations on a concrete service S2 that shows a different interaction protocol. The sequence of calls from the composition is intercepted by a proxy that passes it to an adapter. This last one, by interpreting the mapping script, translates it into a sequence of calls on S2 and returns the results back to the proxy. The runtime infrastructure shown in the figure is part of the *SCENE* framework [17] that, thanks to the intermediation of proxies, supports dynamic binding of services to a certain service composition. *SCENE* has been originally designed under the hypothesis that all services would exhibit identical interfaces or protocols. In our extension this limiting hypothesis is overcome by the introduction of the adapter, a piece of software integrated in *SCENE* proxy that supports mismatches solution by interpreting some mapping scripts. These scripts can be manually provided by a system integrator, as described in [4], or can be automatically generated by the proxy when the service to be bound to the composition is selected. Next section provides details about automatic generation of mapping scripts.

5 Generation of Adaptation Scripts for Protocol-Level Mismatches

In previous section we outlined how adaptation takes place once a mapping script is provided. Building the script may be a hard task for humans and in [5] we proposed an automated solution limited to interface level mismatches. In this section we focus on protocol-level mismatches and on how to build, possibly in an automatic way, a suitable adaptation script.

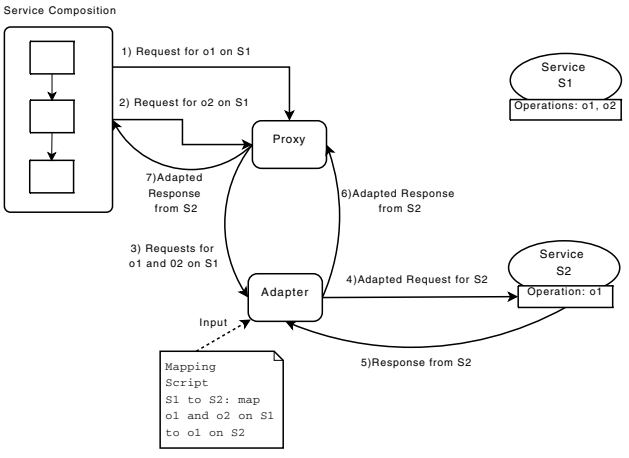

Fig. 1. The adaptation process

5.1 Problem Statement

We assume to know for each service the following information:

- A table which associates to each service operation its input and output parameters. For the example of Section 3 this information is represented by Tables 1 and 2.
- A description of the LTS model associated to the service. This is used to derive the order in which service operations may be invoked. A human-readable version of the LTS models of *Flickr* and *Picasa* is shown in Figures 2 and 3.

We make the hypothesis that both these pieces of information come as a service description that can be accessed and interpreted by both a human or a machine service requestor as *facets* (see [18] for details). The protocol mapping between an abstract and a concrete service assumes that two compatibility relationships have been previously defined. The first relationship states the *compatibility between states* of two LTS models. The second relationship concerns the *compatibility between name and data* associated to some operation $o_{abs} \in O_{abs}$ in the abstract service and those associated to some operation $o'_{conc} \in O_{conc}$ in the concrete service. For the sake of simplicity, we assume in this paper that compatible states, operation names, and data have been already identified someway (for instance, as described in [5]). For this reason, the triple $\langle name, type, value \rangle$ fully characterizing each datum is synthesized here only by the $name$ element.

Given these definitions and considering the LTS models P_{abs} and P_{conc}, referring, respectively, to an abstract and concrete service, we say that, given a sequence of operations in P_{abs} (let us call it seq_{abs}), leading from a state s^i_{abs} to some state s^f_{abs}, this can be *substitutable* by another sequence of operations in P_{conc}, seq_{conc}, provided that:

1. seq_{conc} starts from a state s^i_{conc} compatible with s^i_{abs} and ends into a state s^f_{conc} compatible with s^f_{abs}. Note that LTSs may be non-deterministic: in this case the

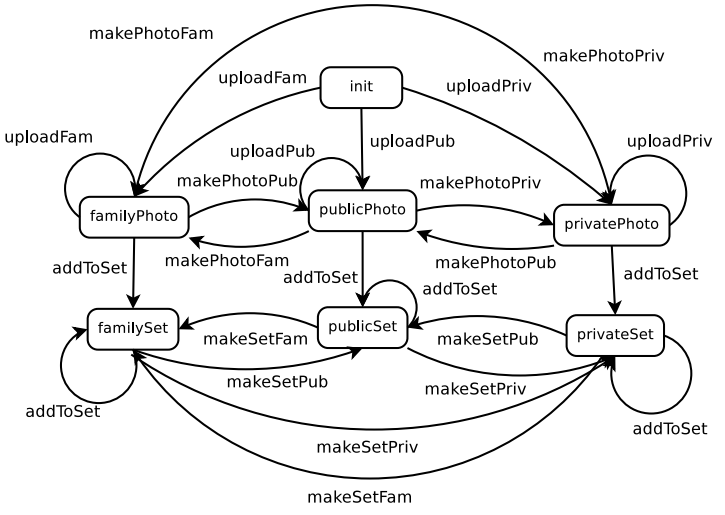

Fig. 2. A representation of the *Flickr* protocol

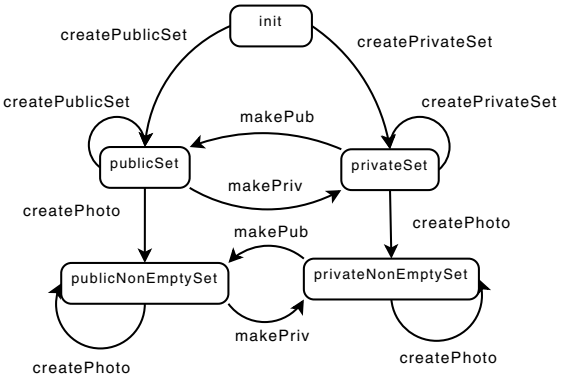

Fig. 3. A representation of the *Picasa* protocol

constraint is that at least one of the ending states $s^{f_1}_{conc} \dots s^{f_a}_{conc}$ of the concrete service is compatible to one of the ending states $s^{f_1}_{abs} \dots s^{f_b}_{abs}$ of the abstract service. From now on we will assume, without loss of generality, that both the LTSs are deterministic.

2. For all operations of seq_{conc}, all data parameters are compatible with those appearing in seq_{abs}.

On this basis we can build a reasoning mechanism that, given some $seq_{abs} = o^1_{abs} \dots o^n_{abs}$ returns a sequence of operations $seq_{conc} = o^1_{conc} \dots o^m_{conc}$ that can replace the first one according to the substitution relationship defined above. We use two different reasoning strategies for identifying seq_{conc}, depending on whether the composition execution

environment supports a synchronous or an asynchronous request-reply semantics for operation calls.

The synchronous semantics requires that in a sequence of operation calls not only the operations are called in the required sequence, but also each operation call cannot be performed before the previous one has returned its foreseen result. An example of this semantics is offered by a BPEL *sequence* block. This mandates that the activities it contains should be executed sequentially.

The asynchronous semantics does not prevent the execution of an operation call even if the previous one has not returned the corresponding value yet, unless there is an explicit dependency between the two in terms of input parameters required by the operation to be started and output parameters produced by the previous operation. Using again an example from BPEL, the asynchronous semantics can be mapped on a *flow* block containing various invoke activities together with the corresponding receives. In this case the BPEL executor interprets the flow block by spanning an independent thread for each activity, still ensuring that each receive statement will be performed after the corresponding invoke, and, if dependent invokes are present, that their execution is properly ordered as well.

The strategies adopted by the reasoning mechanism are then the following:

- *Strategy 1 - Synchronous request-reply semantics*. Given the initial state s^i_{abs} of the abstract sequence seq_{abs} and the corresponding compatible state in the concrete LTS model s^i_{conc}, each transition departing from s^i_{conc} is considered as a candidate to be the o^1_{conc} operation in seq_{conc} provided that all the data it requires as input can be available at the time it will be executed, and the data o^1_{conc} produces as output include those expected by the consumer of o^1_{abs}, if any. The same line of reasoning is applied starting from any s^x_{conc} until the state s^f_{conc} is reached.
  From the runtime perspective, this results in the fact that an operation $o_{abs} \in seq_{abs}$ can be invoked only if the previous one in the sequence has been completed, that is, the corresponding counterpart in the concrete service has returned the proper value.
- *Strategy 2 - Asynchronous request-reply semantics*. Given the initial state s^i_{abs} of the abstract sequence seq_{abs}, the corresponding compatible state in the concrete LTS model s^i_{conc}, and the final state s^f_{abs}, the transitions $o^1_{conc}...o^m_{conc}$ are considered as possible candidate operations for seq_{conc} provided that:
  1. all the data each operation in seq_{conc} requires as input are available at the time the operation is executed;
  2. all the data expected as output by operations in seq_{abs} will be produced by the operations in seq_{conc} by the time s^f_{conc} is reached.

  At runtime, this implies that, given an invoked operation $o_{abs} \in seq_{abs}$ which returns some data, the next operation in sequence can be invoked without necessarily waiting that the result of o_{abs} has been provided. Consequently, any kind of binding can be established from some operations $o^1_{abs}...o^n_{abs} \in seq_{abs}$ into one or more operations in seq_{conc}, since, for every $x \in [1, n]$ the service consumer may invoke an operation o^{x+1}_{abs} even if o^x_{abs} has not returned yet. Of course this statement is valid if o^{x+1}_{abs} does not require any of the return parameter of o^x_{abs} as input.

Intuitively, the synchronous semantics limits the kind of mismatches for which a solution can be found. In this situation, many to one and many to many bindings can be

treated in the general case only if operations involved in the mismatch require no return values. Consider for instance the example in Section 3. Given the trace: *uploadPriv*, *addToSet*, *makeSetPub* departing from the *init* state on *Flickr*, there is no possibility to build a mapping script allowing for the usage of *Picasa* in the synchronous case. In fact, applying the synchronous request-reply semantics reasoning schema, the first operation to be invoked on *Picasa* should accept as input a set of parameters included in those provided to *uploadPriv* on *Flickr*, and should return at least all the parameters expected in return by the same *Flickr* operation. Since all the operations outgoing from *init* on *Picasa* require as input a *albumName* and this datum is not provided by *uploadPriv*, no operation on *Picasa* is a valid candidate and, consequently, it is impossible to build a mapping script.

In the case the asynchronous request-reply semantics schema is applied, a mapping can be identified. In fact, *addToSet* in seq_{abs} can be invoked even if the operation call *uploadPriv* has not produced its return value yet as it does not have a direct counterpart in seq_{conc}. After *addToSet* is invoked, *createPublicSet* or *createPrivateSet* in *Picasa* can be invoked as their input parameter (*albumName*) is available. Indeed, both produce a *success* output, which is expected by the service requestor as output of one of the invoked abstract operations. Assuming that *createPublicSet* is chosen for invocation, there are two possible operations candidate for being part of the concrete sequence: *makePriv* and *createPhoto*. Between those *createPhoto* is chosen because it is the only operation that returns the second *success* output, which is expected by the service requestor. This last operation leads *Picasa* into the *publicNonEmpty* state that is compatible with final state of the abstract sequence, that is, *publicSet*.

From the above examples the reader should notice that both strategies are based on the assumption that the substitution is totally transparent to the service consumer, who invokes the abstract service operations, provides input data for those operations and expects some return data from them. The invocations performed to the abstract service operations are translated into invocations to concrete service operations: input data provided by the consumer are used as input for the invoked concrete service operations and return data provided by the invoked concrete operations are returned to the consumer as needed. Any input parameter provided by the consumer is stored and can be used as input for a concrete operation requiring it. When this happens the parameter is removed from the storage. The same line of reasoning is valid for output parameters, if we consider that they are provided by the concrete service and are returned to the service consumer.

5.2 Implementation and Practical Issues

The reasoning mechanism has been formulated using the linear temporal logic language TRIO [19]. Our model features some application-independent TRIO formulas that represent the reasoning strategies as expressed in the previous section, and some application-dependent formulas, which represent the interfaces and protocols of the abstract and concrete services.

Given this model and an operations sequence seq_{abs}, the approach formulates the problem of finding a substitutable operation sequence seq_{conc}. If this sequence exists, a mapping script is generated. The script is executed by the adapter that, as shown in Figure 1, receives the sequence of invocations that the service consumer expects to perform and transforms them into invocations suitable for the concrete service.

We have chosen to implement the model of the reasoning mechanism using *Zot*[6], an agile and easily extensible bounded model- and satisfiability-checker. In general, *Zot* returns a history (i.e., an execution trace of the specified system) which satisfies the given model. The history contains a finite number of steps, each one consisting of a possible configuration of the system.

In our approach the history returned by *Zot* is a mapping script that is then passed as input to the adapter (see Section 4 for details). Each history step contains the state in which each one of the analyzed LTS (the ones of the abstract and concrete services) is, the operations in seq_{abs} and in seq_{conc} that should be invoked in that step, and the exchanged data, if any. In the current implementation, we make the hypothesis that at most one operation in seq_{abs} and at most one in seq_{conc} can be executed at each history step.

Consider again the operations *uploadPriv*, *addToSet*, *makeSetPub* as seq_{abs} departing from the *init* state on *Flickr*. Let us assume an asynchronous semantics and specify as compatible the *init* states of the two services and the states *publicSet* of *Flickr* and *publicNonEmptySet* of *Picasa*. In this case, a possible history returned by *Zot* is reported in Table 3. In the first two steps the history only reports invocations on *Flickr*. This means that the adapter only expects to receive invocations from the service consumer and to keep trace of provided inputs and required outputs. On step 3 there are enough data to invoke the operation *createPublicSet* on *Picasa*. The adapter performs the invocation on the concrete service, uses as input for that invocation the *albumName* stored in memory, and removes the parameter from storage. The *success* value returned by this operation is forwarded to the service consumer. On step 4 the history reports again an invocation on *Flickr*. In this case the adapter behaves exactly as in steps 1 and 2. Finally on step 5 the history mandates the invocation on *Picasa* of the operation *createPhoto* and on step 6 *Flickr* is in a state *publicSet*, considered final for the considered sequence and *Picasa* is in a state compatible to *publicSet*.

6 Evaluation

The experiments were conducted to prove the effectiveness in solving protocol level mismatches and the performance of the approach both as an interactive and on-line solution to determine feasible mappings[7]. In particular, we conducted two classes of experiments.

- We ran experiments with *Flickr* and *Picasa* trying to map various abstract sequences into some concrete ones in order to see if the approach was behaving as expected in terms of the identification of correct mappings.

[6] *Zot* can be downloaded from http://home.dei.polimi.it/pradella

[7] The input set of experiments is available at http://home.dei.polimi.it/cavallaro/evaluation-experimentsInputs.zip

Table 3. An history generated for the seq_{abs} = *uploadPriv*, *addToSet*, *makeSetPub*

Step	*History Content*
1	*Flickr*State = init; *Flickr*Invoke = uploadPriv *Flickr*Input = photo, photoName; *Flickr*Output = success *Picasa*State = init
2	FlickrState = privatePhoto; *Flickr*Invoke = addToSet *Flickr*Input = albumName, photoName; *Flickr*Output = success *Picasa*State = init
3	*Flickr*State = privateSet *Picasa*State = init; *Picasa*Invoke = createPublicSet *Picasa*Input = albumName; *Picasa*Output = success
4	*Flickr*State = privateSet; *Flickr*Invoke = makeSetPub *Flickr*Input = albumName *Picasa*State = publicSet
5	*Flickr*State = publicSet *Picasa*State = publicSet; *Picasa*Invoke = createPhoto *Picasa*Input = photo, photoName, albumName; *Picasa*Output = success
6	*Flickr*State = publicSet *Picasa*State = publicNonEmptySet

– We compared the performance of our approach with the one shown by a similar approach found in the literature [15].

All the experiments had the goal of exploring the possibility for our tool to derive (whenever possible) correct mappings between an abstract and a concrete service. The experiments were conducted on a 2.5 Ghz Intel Core2 duo machine, equipped with 4 GBytes of memory, running Linux. The Common Lisp compiler used for running *Zot* was SBCL, version 1.0.18.

The main inputs used in each experiment have been: a) the LTSs of the abstract service and the candidate concrete service b) the associations between service operations and their inputs and outputs; c) the compatibility relationship between the operation names and parameters of the abstract and concrete services; and d) a possible seq_{abs}. The results obtained by the experiments have been a possible seq_{conc} in the cases this could have been identified by the tool as well as information about the time needed by the tool to produce a result or to signal the impossibility of producing it.

As additional input, since *Zot* is based on a SAT-solver, it is necessary to set the size k of the periodic temporal structure on which the verification is performed. In this case, all the periodic behaviors of the system, with period up to k are considered by the tool. The identification of a proper value for k is always a critical issue when exploiting a SAT-solver. High values for k usually imply long execution times for the tool while small values may result in the fact that the tool is not able to find a solution that would have been identified if the considered temporal structure was longer. Our approach is essentially based on constructing the product of the abstract and concrete LTSs, hence the upper bound for non-cyclic behaviors is $ns_{abs} \cdot ns_{conc} - 1$, where ns_{abs} and n_{conc} are the number of states of the LTS models of the abstract and concrete

Table 4. Results for the experiments on examples in Section 3

			Time (s)			
Uploaded photos	Sequence length	ns	ns	$2ns$	$3ns$	$4ns$
1	3	12	0.59	1.87	3.72	5.80
2	6	12	0.59	1.94	4.06	7.32
3	9	12	0.55	1.81	4.14	7.35
4	13	12	0.55	1.86	4.82	7.72

services, respectively. In practice, we empirically found that in most of the cases a good estimate for k is $ns = ns_{abs} + ns_{conc}$. With $k = 2ns$ we were able to find solutions for every considered case. Therefore the algorithm first tries with $k = ns$, then considers $k = 2ns$, and so on, keeping $ns_{abs} \cdot ns_{conc}$ as an upper bound. In the experiments we considered four possible values for k: $2ns$, $3ns$, and $4ns$, to see how the tool speed is affected by increasing bounds.

Experiments with Flickr and Picasa. We ran the tool starting from the *Flickr* abstract sequence we have used through this paper. Moreover, we have complicated it considering the case in which up to 4 pictures are uploaded (this results in the fact that the operations *uploadPriv* and *addToSet* are called more than one time. The results are reported in Table 4. We started with a bound $k = ns = 12$. In the first two cases reported in the table (upload of one and two pictures) we succeeded in determining a sequence with ns, while in the last two cases we needed to use $k = 2ns$. The overheads introduced to produce a working mapping script are between 0.59 and 1.86 seconds. This makes the approach suitable for both on-line and off-line use at least in this specific case. The histories produced by *Zot* were analyzed by a human to prove their correctness and were executed by the adapter as mapping scripts. The performed tests succeeded in using *Picasa* in place of *Flickr*.

Comparison with [15]. We compared our technique with the one presented in [15] and summarized in Section 2. The tool is called Dinapter, and its package contains several examples of abstract and concrete services. We took some of the most significant ones and used them both with Dinapter and *Zot*.

In the original example, the tested services were all described using abstract BPEL. They contain branches, loops and non-determinism. In order to use them with our tool we translated the abstract BPEL description into LTS using the following criteria:

- For what concern the BPEL descriptions representing sequences of calls, we considered invoke activities as operation invocations, and receive activities associated with invocations and featuring parameters as responses to the invoked operations.
- For BPEL description representing service interfaces, we considered receive activities as invocation expected by the service, and invoke operations featuring parameters and associated with the receives as issued responses.
- We considered those activities included in a BPEL *sequence* block as having a synchronous semantics.

Table 5. Results of the comparison with [15]

		[15]	Our approach (Time (s))			
Example name	ns	Time (s)	ns	$2ns$	$3ns$	$4ns$
e001-ftp-tiny	6	1.4	0.06	0.34	0.54	0.9
e002-ftp-small	8	30.65	0.11	0.53	0.95	1.54
e002c-ftp-small	7	37.15	0.12	0.39	0.81	1.30
e003-ftp-full	8	Out of memory	0.17	0.26	0.48	0.75
e004-wich-Pick	10	45.10	0.75	2.37	4.81	8.60
e005-start-Switch	8	51.05	0.53	1.62	2.87	4.45
e010-Pick-Pick	12	6.01	0.64	2.09	3.51	7.03
e013-deceptive-Pick	12	54.90	0.68	2.01	3.47	6.95
e017-2Switch-2Pick-carry	10	Out of memory	0.34	0.91	1.92	3.49
vod-1	8	14.41	0.09	0.23	0.71	1.10

The results of the comparison are reported in Table 5. In each row, the name of the example taken from the test set bundled with Dinapter is reported. The time needed to run Dinapter (third column) is the one we calculated by executing the tool on our reference machine. The other times in the last four columns are those referred to our tool with the temporal structure bound k set to the first four multiples of ns, i.e., the sum of the abstract and concrete LTSs states.

Our approach was able to find a solution in every case with the bound estimated as ns and with an execution time shorter than 1 second (clearly the time increases for higher values of the bound). This, again, is promising for on-line use of the tool. Moreover, our approach outperformed Dinapter that in some cases has not been able to terminate with success because of out of memory problems. The output sequences produced by *Zot* were inspected by a human to verify correctness and, in those cases in which Dinapter was able to produce a result, were compared with those produced by Dinapter and found out to be equivalent.

7 Conclusion

In this work we presented an approach to identify an interaction protocol mapping between compatible conversational services. The mapping is deduced by using *Zot*, a recent, efficient model checker based on a SAT-solver.

We validated our technique by considering two real-life services, *Flickr* and *Picasa*, obtaining both correct protocol mappings between the two and good performance. Moreover, we compared our approach with Dinapter [15] on some significant cases that have been made available together with this last tool. *Zot* outperformed Dinapter in all cases, with times suitable for on-line application of the technique. The research work is currently ongoing and disregards some important aspects that need to be considered for successful service replacement. Currently we analyzed only services featuring conversations that can be represented by LTSs, while some real world cases need more powerful formalisms (e.g. services featuring branches executed in parallel, services featuring

not only a conversational state but also an internal state). Finally services are usually invoked in complex processes that may feature a state or transactional support. Consequently service substitution may require house keeping work of the running processes. Thus, as future work we plan to extend our approach to allow consistent substitution of stateful and transactional services.

References

1. Baresi, L., Nitto, E.D., Ghezzi, C.: Toward open-world software: Issue and challenges. IEEE Computer 39(10), 36–43 (2006)
2. Verma, K., Gomadam, K., Sheth, A.P., Miller, J.A., Wu, Z.: The METEOR-S approach for configuring and executing dynamic web processes. University of Georgia, Athens, Tech. Rep. (June 2005)
3. Antonellis, V.D., Melchiori, M., Santis, L.D., Mecella, M., Mussi, E., Pernici, B., Plebani, P.: A layered architecture for flexible web service invocation. Software Practice and Experience 36(2), 191–223 (2006)
4. Cavallaro, L., Di Nitto, E.: An approach to adapt service requests to actual service interfaces. In: Proceedings of SEAMS (2008)
5. Cavallaro, L., Ripa, G., Zuccalà, M.: Adapting service requests to actual service interfaces through semantic annotations. In: Proceedings of PESOS (2009)
6. Moser, O., Rosenberg, F., Dustdar, S.: Non-intrusive monitoring and service adaptation for WS-BPEL. In: Proceedings of WWW (2008)
7. Mateescu, R., Poizat, P., Salaün, G.: Adaptation of service protocols using process algebra and on-the-fly reduction techniques. In: Bouguettaya, A., Krueger, I., Margaria, T. (eds.) ICSOC 2008. LNCS, vol. 5364, pp. 84–99. Springer, Heidelberg (2008)
8. Dumas, M., Spork, M., Wang, K.: Adapt or perish: Algebra and visual notation for service interface adaptation. In: Dustdar, S., Fiadeiro, J.L., Sheth, A.P. (eds.) BPM 2006. LNCS, vol. 4102, pp. 65–80. Springer, Heidelberg (2006)
9. Nezhad, H.R.M., Benatallah, B., Martens, A., Curbera, F., Casati, F.: Semi-automated adaptation of service interactions. In: Proceedings of WWW 2007 (2007)
10. Drumm, C.: Improving schema mapping by exploiting domain knowledge. Ph.D. dissertation, Universitat Karlsruhe, Fakultat fur Informatik (2008)
11. Fredj, M., Georgantas, N., Issarny, V., Zarras, A.: Dynamic service substitution in service-oriented architectures. In: Proceedings of SERVICES (2008)
12. Brogi, A., Popescu, R.: Automated generation of BPEL adapters. In: Dan, A., Lamersdorf, W. (eds.) ICSOC 2006. LNCS, vol. 4294, pp. 27–39. Springer, Heidelberg (2006)
13. WS-BPEL specification, `http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsbpel`
14. van der Aalst, W.M.P., ter Hofstede, A.H.M.: Yawl: yet another workflow language. Information Systems 30(4), 245–275 (2005)
15. Martìn, J.A., Pimentel, E.: Automatic generation of adaptation contracts. In: Proceedings of FOCLASA (2008)
16. Fielding, R.T.: Architectural styles and the design of network-based software architectures. Ph.D. dissertation, chair-Taylor, Richard N (2000)

17. Colombo, M., Di Nitto, E., Mauri, M.: Scene: A service composition execution environment supporting dynamic changes disciplined through rules. In: Dan, A., Lamersdorf, W. (eds.) ICSOC 2006. LNCS, vol. 4294, pp. 191–202. Springer, Heidelberg (2006)
18. Colombo, M., Di Nitto, E., Penta, M.D., Distante, D., Zuccalà, M.: Speaking a common language: A conceptual model for describing service-oriented systems. In: Benatallah, B., Casati, F., Traverso, P. (eds.) ICSOC 2005. LNCS, vol. 3826, pp. 48–60. Springer, Heidelberg (2005)
19. Ghezzi, C., Mandrioli, D., Morzenti, A.: Trio: A logic language for executable specifications of real-time systems. Journal of Systems and Software 12(2) (1990)