HAL
open science

Synthesizing adapters for conversational web-services from their WSDL interface

Luca Cavallaro, Elisabetta Di Nitto, Patrizio Pelliccione, Matteo Pradella, Massimo Tivoli

To cite this version:

Luca Cavallaro, Elisabetta Di Nitto, Patrizio Pelliccione, Matteo Pradella, Massimo Tivoli. Synthesizing adapters for conversational web-services from their WSDL interface. In ICSE 2010 SEAMS: Workshop on Software Engineering for Adaptive and Self-Managing Systems, 2010, Capetown, South Africa. pp.104-113. inria-00512413

HAL Id: inria-00512413

https://inria.hal.science/inria-00512413v1

Submitted on 22 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.

Synthesizing adapters for conversational web-services from their WSDL interface*

Luca Cavallaro[1], Elisabetta Di Nitto[1], Patrizio Pelliccione[2], Matteo Pradella[3], Massimo Tivoli[2]
[1] Politecnico di Milano, DEI, Piazza L. Da Vinci, 32, 20133 Milano, Italy
{cavallaro, dinitto}@elet.polimi.it
[2] Università dell'Aquila, Dipartimento di Informatica, Via Vetoio, L'Aquila, Italy
{patrizio.pelliccione, massimo.tivoli}@di.univaq.it
[3] CNR IEIIT-MI, via Golgi 42, 20133 Milano, Italy
pradella@elet.polimi.it

ABSTRACT

Service-oriented applications are typically built out of existing web-services (WSs) possibly made available by third party vendors. This requires that the application has to be able to evolve when the composing WSs are not anymore available or when new, more useful ones, are published. In this setting, an important problem is to understand how to use WSs showing an interface that differs from the one the application has been built to. The problem becomes even more complex when we consider conversational WSs, i.e., WSs that expose operations that have Input/Output (I/O) data dependencies among them. This paper presents a complete development methodology to the automatic synthesis of adapters for conversational WSs starting from their WSDL interface. The result is a tool-supported methodology that takes as input the WSDL of a pair of services and automatically builds a script that maps a sequence of operation invocations on a "WS to be replaced" into an equivalent sequence of operation invocations on the "replacing WS". The overall approach is presented by applying it to two existing WSs that realize two distinct, but equivalent, search engines for lyric music.

Categories and Subject Descriptors

D.2.10 [**Software Engineering**]: Design; D.2.5 [**Software Engineering**]: Testing and Debugging; D.2.2 [**Software Engineering**]: Design Tools and Techniques—*Computer-aided software engineering (CASE)*; H.3.5 [**Information Storage and Retrieval**]: On-line Information Services—*Web-based services*

General Terms

Design, Experimentation, Theory, Verification.

*This work is partly supported by the EU project CONNECT (http://connect-forever.eu/) No 231167 of the FET - FP7 program and the Italian Government under the project PRIN 2007 D-ASAP (2007XKEHFA). We would like to thank also Antonia Bertolino and Paola Inverardi for joint efforts on the research concerning `StrawBerry`.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SEAMS '10, May 2-8, 2010, Cape Town, South Africa
Copyright 2010 ACM 978-1-60558-971-8 ...$10.00.

Keywords

Service composition, Service adapters, Web-services, Automatic synthesis, Testing, Behavior protocols.

1. INTRODUCTION

Service Oriented Architectures (SOAs) are a flexible set of design principles that promote interoperability among loosely coupled services that can be used within multiple business domains. Web-based applications are typically *composed* web-services (WSs) built out of existing WSs possibly made available by third party vendors. This opens a series of new scenarios that are unimaginable under the closed world assumption. On one hand, an organization cannot control each part of the application and, hence, failures and service unavailability should be taken into account. On the other hand, new interesting WSs can become available thus enabling new features or providing equivalent services with better quality. Therefore, frameworks that are able to support the evolution of a composition of WSs, by letting the web-based application deal with any alternative service, become of extreme importance.

Most of the frameworks proposed in the recent years make the assumption that all the semantically equivalent WSs have an agreement on the interface [16, 27]. In the practice this assumption turns out to be unfounded. Moreover, when dealing with stateless WSs all the operations are independent and adaptation has to deal with only operation names and data structures. The situation becomes more complicated when dealing with *conversational WSs*, i.e., WSs that expose operations that have Input/Output (I/O) data dependencies among them. In fact in this case, the composition must take into account the correct sequences of operation invocations, i.e., the *behavior protocol*.

Some of the authors of this paper described, in a previous work [14], an approach for automatically enabling the dynamic replacement of conversational services. In this paper, we build on this previous work by presenting a complete development methodology to the automatic synthesis of adapters for conversational WSs starting from their WSDL interface. The contribution of this paper is to add to the previous work the possibility to describe WSs only through their WSDL. This removes the often unrealistic assumption, made in [14], that the services are described in terms of a model of their behavior protocol, beyond a description of their signature (e.g., WSDL). In this paper we show that integrating the approach in [14] with `StrawBerry` [12] it is possible to derive such a model from the WSDL interface of a WS.

The result of this integration is a tool-supported methodology that takes as input the WSDL of a pair of services and automatically builds an adapter that realizes a mapping between a sequence of

operation invocations on a WS to be replaced and an equivalent sequence of operation invocations on the replacing WS.

The overall approach is described and validated by applying it to two existing WSs that realize two distinct, but equivalent, lyric search engines.

The paper is structured as follows: Section 2 presents an overview of the approach by outlining the previous work on which the approach it is based on. This section presents also a motivating example that will be used as running example through out the remaining of the paper. Section 3 presents the approach. Section 4 discusses related work and, finally, Section 5 concludes the paper and outlines future research directions.

2. OVERVIEW OF THE APPROACH

This section is organized as follows: Section 2.1 presents a motivating example that will be used as running example in the remaining of the paper, Section 2.2 outlines `StrawBerry`, and finally Section 2.3 outlines the approach presented in [14].

2.1 Motivating example

To motivate our methodology and to illustrate it through a stepwise description (see Section 3), we use an example concerning two existing conversational WSs available on the Internet. These two WSs realize two lyric search engines. One is called `ChartLyrics` [1], the other is called `LyricWiki` [2].

`ChartLyrics` is a lyrics database sorted by artists or songs. The `ChartLyrics` API uses either a SOAP or REST interface to allow users and developers to access the database. For our purposes, we consider the SOAP interface. The WSDL of the `ChartLyrics` SOAP interface [3] provides three operations: (i) `SearchLyric` to search the available lyrics, (ii) `SearchLyricText` to search a song by means of some text within an available lyric text, and (iii) `GetLyric` to retrieve the searched lyric.

`LyricWiki` is a free site which is a source where anyone can go to get reliable lyrics for any song from any artist. It uses only a SOAP interface. The WSDL of `LyricWiki` [4] provides several operations. Among the others, there are five operations of interest for our purposes: (i) `searchSongs` to search for a possible song on `LyricWiki` and get up to ten close matches, (ii) `checkSongExists` to check if a song exists in the `LyricWiki` database yet, (iii) `getSong` to get the lyrics for a searched `LyricWiki` song with the exact artist and song match, (iv) `searchArtists` to search for a possible artist by name and return up to ten close matches, and (v) `getArtist` to get the entire discography for a searched artist. For the sake of presentation, in the following, we consider a version of the WSDL of `LyricWiki` modified in order to both take into account only the above five operations and increase the meaningfulness of the example.

As it will be described in detail in Section 3.1, to get a lyric by using `ChartLyrics`, a client can exploit the following sequence of operation invocations: `SearchLyric`, `GetLyric`. Whereas to get a lyric by using `LyricWiki`, a right sequence of operation invocations is, e.g., the following: `checkSongExists`, `searchSongs`, `getSong`.

If `LyricWiki` were part of a web application realized through a service composition, it could happen that, in certain circumstances, it would need to be replaced by `ChartLyrics` or by any other specialized search engine. This could happen, for instance, to accommodate the preferences of users having their preferred engine, or to handle the cases when `LyricWiki` is unavailable for any problem. The developer could think to code, by hand, the instructions to deal with any possible engine and its replacement. However, this approach does not allow the application to deal with unforeseen search engines. As it will be described in detail in Section 3.3, a better solution is to build a mapping mechanism that dynamically handles the mismatches by automatically synthesizing a behavior protocol mapping script. The adaptation realized by the synthesized mapping script could state, e.g., that the sequence of `LyricWiki` operations `checkSongExists`, `searchSongs`, `getSong` maps on the sequence of `ChartLyrics` operations `SearchLyric`, `GetLyric`. This mechanism would make the application able to deal with any unforeseen lyric search engine.

2.2 The StrawBerry approach

`StrawBerry` derives from the WSDL of a WS, in automated way, a partial ordering relation among the invocations of the different WSDL operations, that we represent as an automaton. This automaton, called *Behavior Protocol automaton*, models the interaction protocol that a client has to follow in order to correctly interact with the WS. This automaton explicitly models also the information that has to be passed to the WS operations. More precisely, the states of the behavior protocol automaton are WS execution states and the transitions, labeled with operation names plus I/O data, model possible operation invocations from the client of the WS. The behavior protocol is obtained through synthesis and testing stages. The synthesis stage is driven by data type analysis, through which we obtain a preliminary dependencies automaton, that can be optimized by means of heuristics. Once synthesized, this dependencies automaton is validated through testing against the WS to verify conformance, and finally transformed into an automaton defining the behavior protocol.

`StrawBerry` is a black-box and extra-procedural method. It is black-box since it takes into account only the WSDL of the WS. It is extra-procedural since it focuses on synthesizing a model of the behavior that is assumed when interacting with the WS from outside, as opposed to intra-procedural methods that synthesize a model of the implementation logic of the single WS operations [20, 28, 29].

Figure 1 graphically represents `StrawBerry` as a process split in five main activities.

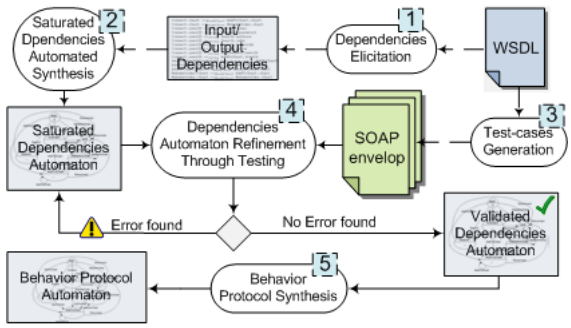

Figure 1: Overview of the `StrawBerry` method

The *Dependencies Elicitation* activity elicits data dependencies between the I/O parameters of the operations defined in the WSDL. A dependency is recorded whenever the type of the output of an operation matches with the type of the input of another operation. The match is syntactic. The elicited set of I/O dependencies may

[1] http://www.chartlyrics.com/api.aspx
[2] http://lyrics.wikia.com/Main_Page
[3] http://api.chartlyrics.com/apiv1.asmx?WSDL
[4] http://lyrics.wikia.com/server.php?wsdl

be optimized under some heuristics [12]. They are general since they concern the syntactic characteristics of a generic WSDL. The elicited set of I/O dependencies (see the *Input/Output Dependencies* artifact shown in Figure 1) is used for constructing a data-flow model (see the *Saturated Dependencies Automaton Synthesis* activity and the *Saturated Dependencies Automaton* artifact shown in Figure 1) where each node stores data dependencies that concern the output parameters of a specific operation and directed arcs are used to model syntactic matches between output parameters of an operation and input parameters of another operation. This model is completed by applying a *saturation rule*. This rule adds new dependencies that model the possibility for the client to invoke a WS operation by directly providing its input parameters. The resulting automaton is then validated against the implementation of the WS through testing (see *Dependencies Automaton Refinement Through Testing* activity shown in Figure 1).

The testing phase takes as input the SOAP messages produced by the *Test-cases generation* activity. The latter, driven by coverage criteria, automatically derives a suite of test cases (i.e., SOAP envelop messages). For this purpose, we use the WS-TAXI [10] tool that takes as input the WSDL of a WS and automatically produces the SOAP envelop messages ready for execution. Note that testing is used here in opposite way with respect to the usual model-based testing (MBT) practice [25]. In fact, in MBT the automaton is used as an oracle, and testing aims at checking whether the implementation conforms to it. In `StrawBerry` instead, tests are generated from the WSDL and aim at validating whether the synthesized automaton is a correct data-flow abstraction of the service implementation. Intuitively, we can say that `StrawBerry` tests if the model conforms to the implementation. Testing is used to refine the syntactic dependencies by discovering those that are semantically wrong. By construction, the inferred set of dependencies is syntactically correct. However, it might not be correct semantically since it may contain false positives (e.g., a string parameter used as a generic attribute is matched with another string parameter that is a unique key). If during the testing phase an error is found, this means that the automaton must be refined since the set of I/O dependencies contains false dependencies.

Once the testing phase is successfully terminated, the final automaton models, following a data-flow paradigm, the set of validated "chains" of data dependencies. `StrawBerry` terminates by transforming this data-flow model into a control-flow model (see the *Behavior Protocol Synthesis* activity in Figure 1). This is another kind of automaton whose nodes are WS execution states and whose transitions, labeled with operation names plus I/O data, model the possible operation invocations from the client to the WS.

2.3 Automatic replacement of conversational services

The approach presented in [14] enables service adaptation through the definition of proper *mapping scripts*. These scripts associate the sequences of operations that the client is assuming to invoke on the expected WS into the corresponding sequences made available by the WS that will be actually used. Such mapping scripts are then interpreted by *adapters* that intercept all service requests issued by the client and transform them into the requests the services are able to fulfill. Figure 2 shows the placement of adapters into the infrastructure architecture and highlights their nature of intermediaries.

Mapping scripts are automatically derived given the following information about all potential candidate services:

- The WSDL interfaces where input and output parameters are associated to each service operation.

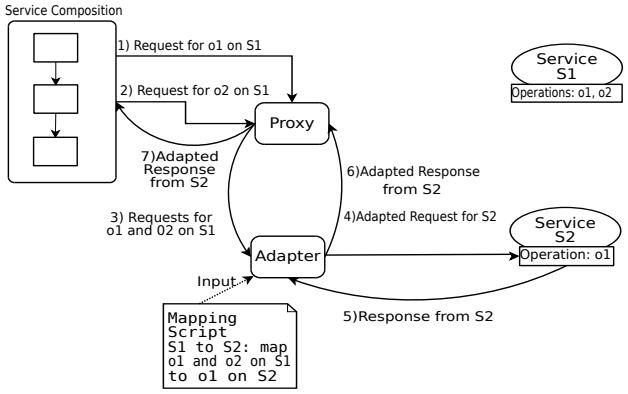

Figure 2: The adaptation runtime infrastructure.

- A description of the behavioral protocol associated to each service given in terms of an automaton.

The mapping between an expected and an actual WS assumes that two compatibility relationships have been previously defined. The first relationship states the *compatibility between states* of two automata. The second relationship concerns the *compatibility between names and data* associated to some operation $o_{exp} \in O_{exp}$ in the expected WS and those associated to some operation $o'_{act} \in O_{act}$ in the actual WS. For the sake of simplicity, here we assume that states and operation names and data are compatible if they are called the same way (more sophisticated compatibility relationships can be identified by assuming the existence of a semantic network of concepts as in [15]).

Given these definitions, we say that, given a sequence of operations in the automaton of the expected WS, this can be *substitutable* by another sequence of operations in the automaton of the actual WS if the following conditions hold:

1. The sequence in the actual WS automaton starts and ends in states that are compatible with the initial and final states of the sequence in the expected WS automaton.

2. All data parameters of the operations in the actual WS automaton sequence are compatible with those appearing in the expected WS automaton sequence.

This substitutability definition allows us to build a reasoning mechanism that, given an expected WS sequence, returns a corresponding actual WS sequence. The reasoning mechanism has been formulated using classical linear temporal logic (LTL). Our model features some application-independent formulas that represent the reasoning strategy and some application-dependent formulas, which represent the interfaces and protocols of the expected and actual WSs. Given this model and an operations sequence in the expected WS automaton, the approach formulates the problem of finding a substituting operation sequence in the actual WS automaton. If this sequence exists, a mapping script is generated and instantiated into the adapter.

The tool used to implement the reasoning mechanism is *Zot*[5] [23], an agile and easily extensible bounded model- and satisfiability-checker. Used as a simulator, *Zot* returns an execution trace of the system made of a finite number of steps, each one consisting of one of its possible configurations. Once the expected and

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

actual service protocols have been elicited, they have to be translated into the *Zot* operational format [24]. The latter features an operational description of the protocol automaton for both the expected and the actual service and of the data exchanged by each transition. The translation step is quite straightforward, except for what concerns the invocation semantics relative to each transition that could follow either the synchronous or asynchronous request-reply pattern.

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. This mandates that given an operation sequence composed of o_1 and o_2 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. Considering again an operation sequence composed of o_1 and o_2 once the first operation has been invoked the second can be invoked without waiting for the result of o_1 to be returned, unless there is an explicit dependency between the outputs of o_1 and the inputs of o_2. Our translation selects the asynchronous semantics, if not otherwise specified by the composition designer or no dependency exists between input and output of operations in a sequence.

The trace returned by *Zot* is used to automatically define a mapping script. Each trace step contains the state in which each one of the analyzed automata (the ones of the expected and actual WSs) is, the operations in the expected and actual WS automata that should be invoked in that step, and the exchanged data, if any.

At runtime the replacement of the expected WS with the actual one is totally transparent to the service user that invokes the expected WS operations, provides input data for those operations and expects some return data from them. Such invocations are translated into actual calls to actual WS operations. Any input parameter provided by the consumer is stored and can be used as input for the actual 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 actual WS and are returned to the service consumer.

3. AUTOMATIC SYNTHESIS OF BEHAVIOR PROTOCOL ADAPTERS

This section presents our approach to the automatic synthesis of adapters for conversational web-services from their WSDL interface. The approach is composed of three main steps: the elicitation of behavior protocols performed by using `StrawBerry` (presented in Section 3.1) the integration between `StrawBerry` and the adapters generation phase (presented in Section 3.2) and, finally, the protocol adapters generation (presented in Section 3.3). We present these steps by using the WSs described in Section 2.1 as reference examples.

3.1 Behavior Protocol elicitation using StrawBerry

The SOAP interface of `ChartLyrics` defines, in its WSDL, the following operations:

• `SearchLyric`: to search the available lyrics by providing two strings: the name of the artist and the title of the song. This operation returns, among other information, two unique identifiers, `LyricChecksum` and `LyricId`, that are used to uniquely identify a song.

Input data	Output data

`artist: string;`
`song: string;`
	`ArrayOf:`
	`LyricChecksum: string;`
	`LyricId: int;`
	`SongUrl: string;`
	`ArtistUrl: string;`
	`Artist: string;`
	`Song: string;`
	`SongRank: int;`

• `SearchLyricText`: to search a song by means of some text within an available lyric text. This operation returns the same information as the one retrieved by `SearchLyric`.

Input data	Output data

`lyricText: string;`	`ArrayOf:`
	`LyricChecksum: string;`
	`LyricId: int;`
	`SongUrl: string;`
	`ArtistUrl: string;`
	`Artist: string;`
	`Song: string;`
	`SongRank: int;`

• `GetLyric`: to retrieve a searched lyric by providing its two unique indentifiers. This operation returns information about the lyric including its text (`Lyric`).

Input data	Output data

`lyricId: int;`
`lyricCheckSum: string;`
	`ArrayOf:`
	`LyricSong: string;`
	`LyricArtist: string;`
	`LyricUrl: string;`
	`LyricCovertArtUrl: string;`
	`LyricRank: int;`
	`LyricCorrectUrl: string;`
	`Lyric: string;`

By referring to Figure 1, we show an overview of how `StrawBerry` would process the WSDL of `ChartLyrics`.

Activity 1: Dependencies Elicitation.
This activity is split in two steps. The first step is mandatory and it is the true dependencies elicitation step. The second is optional and performs an optimization through some heuristics. For the sake of brevity, in this paper, we do not discuss the kind of heuristics that the `StrawBerry`'s user can enable to reduce the initial set of elicited data dependencies. For a detailed description of these heuristics, we entirely refer to [12].

Step 1.1, dependency set elicitation: `StrawBerry` automatically derives a "flat" version of the WSDL. This flattening process aims at making the structure of the I/O messages of the WSDL operations explicit with respect to the element types defined in the XML schema of the WSDL. For example, a type that defines an array of items is reduced into the type of a single item in the array. Note that this is a reasonable simplification for the kind of data-flow analysis `StrawBerry` carries on. Starting from the flattened WSDL, `StrawBerry` syntactically matches the type of an output element of an operation with the type of an input element of another operation hence calculating all possible combinations. For example, the following are four possible elicited dependencies:

$$\texttt{SearchLyric.SearchLyricResponse_LyricId} \mapsto_{int} \texttt{GetLyric.GetLyric_lyricId}$$
$$\texttt{SearchLyric.SearchLyricResponse_SongRank} \mapsto_{int} \texttt{GetLyric.GetLyric_lyricId}$$
$$\texttt{SearchLyric.SearchLyricResponse_LyricChecksum} \mapsto_{string} \texttt{GetLyric.GetLyric_lyricCheckSum}$$
$$\texttt{SearchLyric.SearchLyricResponse_LyricChecksum} \mapsto_{string} \texttt{SearchLyric.SearchLyric_song}$$

For instance, $\texttt{SearchLyric.SearchLyricResponse_LyricId} \mapsto_{int} \texttt{GetLyric.GetLyric_lyricId}$ means that, the value of `LyricId`, as output of `SearchLyric`, can be set as input parameter `lyricId` of `GetLyric`.

Given a data dependency, we refer to its left hand-side operation as the *source* operation, and to the right hand-side operation as the *sink* operation. Dependencies are labeled as *certain* or *uncertain*. Initially all derived dependencies are marked as uncertain; as we collect more evidence (which happens via testing or through appli-

cation of heuristics), uncertain dependencies are either eliminated or promoted to certain.

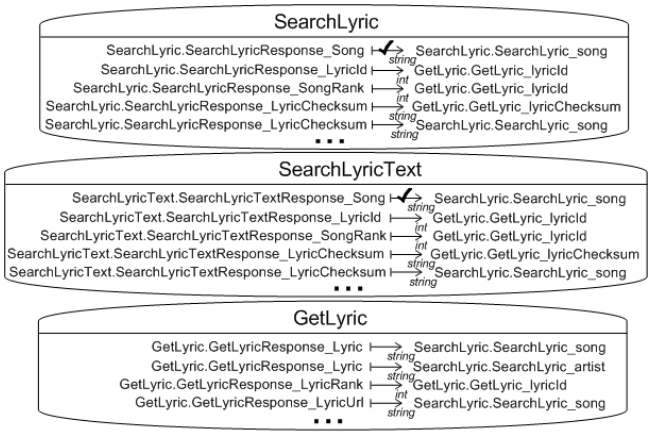

Figure 3: Generated Nodes

Activity 2: Saturated Dependencies Automaton Synthesis.

Step 2.1, node generation: once the data dependencies are elicited, `StrawBerry` synthesizes the dependencies automaton. To do this, for each WSDL operation, that has at least one elicited dependency, `StrawBerry` builds a node.

In Figure 3, we show a portion of the nodes built for `ChartLyrics`. A node stores the name of the operation and the data dependencies defined on its output parameters. In Figure 3, *certain* dependencies are identified by the tick (✓). Certain dependencies, here, are the ones promoted by the application of the heuristics.

Step 2.2, Dependencies automaton synthesis: each arc from a node to another node reflects the data dependencies stored in the source node. The dependencies automaton for `ChartLyrics` is shown in Figure 4. The `start` node and the dotted lines represent the node and the arcs added by the saturation phase explained in the following step.

Step 2.3, Dependencies automaton saturation: for testing purposes we need to take into account also the possibility for the client to directly provide inputs to the WS operations. Thus, we add a new node, `start`. This node stores *uncertain* dependencies conforming to the pattern: $\bigstar \mapsto_T \texttt{Op.p}$ for each sink operation `Op` and for each input parameter `p` of `Op` of type T. The symbol $\bigstar$ denotes a datum directly provided by the client. For space reasons, we do not show the content stored into `start`. According to the dependencies stored into `start`, the saturation step adds an arc from `start` to every other depending node.

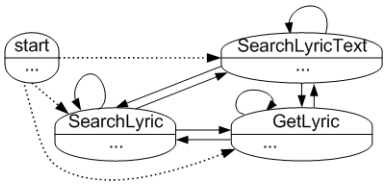

Figure 4: Saturated dependencies automaton

Activity 3: Test-cases Generation.

As said, the only input to `StrawBerry` is a WSDL description. From it, `StrawBerry` derives the black-box test cases that are used in the testing stage (see next activity). Since the test subject is a WS, a test case consists of a SOAP envelope message whose input parameters are filled with appropriate data values. There exist several tools that help to automatically derive such test cases from WSDL, among which soapUI [1] is probably the most popular. `StrawBerry` adopts the WS-TAXI tool [10] that enhances soapUI by fully automating test case derivation. For space limitation, we do not provide all the details of the WS-TAXI functioning, which can be found in [10]. It is worth however to clarify how WS-TAXI deals with input parameter values.

Listing 1 shows an example of a SOAP envelope message produced by WS-TAXI for testing `SearchLyric`. This test case aims at performing a lyric search based on the author and song title criteria. In Listing 1, the `artist` and `song` parameters are randomly generated, which is the default approach of WS-TAXI for string type when no value is available. However, randomly generated string, such as `KOVjot...` below, are not very useful for testing purposes. To overcome this problem, WS-TAXI can derive more meaningful values from a populated database, when available.

Listing 1: Generated SearhLyric SOAP envelope message

```
<soapenv:Envelope
xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:api="http://api.chartlyrics.com/">
  <soapenv:Header/>
  <soapenv:Body xmlns="http://api.chartlyrics.com/">
    <api:SearchLyric>
      <api:artist>KOVjotMZBEfbeynkhtAviBIEs</api:artist>
      <api:song>HJDSFGhfklskjgHFSKJnee</api:song>
    </api:SearchLyric>
  </soapenv:Body>
</soapenv:Envelope>
```

In our approach, it is both necessary and reasonable to assume that, for some of the WSDL input parameters, a set of meaningful values, called an *instance pool* [19] is available. For example, in the case of `SearchLyric`, it is necessary to use a correct value for `artist` and/or `song`. Thus, we assume to have an instance pool of valid `artist/song` values. These instance pools can be reasonably provided by an application user or a domain expert. For instance, it is possible to produce a list of artists probably stored into the `ChartLyrics` database (e.g., some very popular singers or music bands). Although, in general, the instance pool assumption might be strong, in a black-box setting, it is unavoidable since instance pool data are WS-specific and there is no way to infer them.

Wrapping up, if an instance pool is available for some operations, `StrawBerry` exploits this useful piece of information feeding the WS-TAXI database. Concerning the `ChartLyrics` example, the instance pool we use for `SearchLyric` and `SearchLyricText` is shown in Figure 5. Back to Listing 1, the artist parameter can be now taken directly from the instance pool in Figure 5, thus producing more realistic test cases.

Operation	Input Data
SearchLyric	artist: The Cure
SearchLyric	artist: Depeche Mode song: Enjoy the Silence

Operation	Input Data
SearchLyricText	lyricText: But I know that this time I have said too much
SearchLyricText	lyricText: I just can't get enough

Figure 5: Instance pools

Activity 4: Dependencies Automaton Refinement Through Testing.

The goal of this activity is to validate and possibly refine the dependencies automaton against the WS implementation. The test cases are selected so to cover all the dependencies; the oracle is

provided by the WS implementation. Note that since we start from the saturated automaton, the objective of the testing is to prune the false dependencies. Coverage driven test selection in this case fulfills completely the purpose, i.e., we are sure we cannot miss any dependency (contrariwise to the well-known risk of missing functionalities in code coverage testing).

When we invoke the WS, we cannot know in advance which is the expected answer. However, we can always assume that for each test invocation, the WS can either return some output values or answer the request by providing an error message. We refer to the two cases as a *regular answer* and an *error answer*, respectively. The problem we have to face now is that, without analyzing the semantics of the message response it is not possible to distinguish between responses to malformed requests (e.g., a wrong `lyricId`) and negative responses to well-formed requests (e.g., a search of a lyric not contained into the DB). Obviously, it is always possible to define an oracle specific for the considered WS that contains the semantics of errors as can be inferred from the WS documentation. The advantage of this solution is a precise oracle while the disadvantage is that it must be built for each WS. For this reason, in [12] a partial, but general, oracle is proposed. It is based on some heuristics defined on the structure of the output SOAP messages. In this paper, we do not further discuss such a general oracle and we refer to [12] for it.

The testing activity is organized into three steps. `StrawBerry` runs positive tests in the first step and negative tests in the second step. Positive test cases reproduce the elicited data dependencies and are used to reject fake dependencies: if a positive test invocation returns an error answer, `StrawBerry` concludes that the tested dependency does not exist. Negative test cases are instead used to confirm uncertain dependencies: `StrawBerry` provides in input to the sink operation a random test case of the expected type. If this test invocation returns an error answer, then `StrawBerry` concludes that the WS was indeed expecting as input the output produced by the source operation, and it confirms the hypothesized dependency as certain. If uncertain dependencies remain after the two steps, `StrawBerry` resolves the uncertainty by assuming that the hypothesized dependencies do not exist. Intuitively, this is the safest choice, given that at the previous step the invoked operation accepted a random input.

Step 4.1, false dependencies elimination: each *uncertain* dependency in every node is tested. For example, considering the dependency `SearchLyricText.SearchLyricTextResponse_LyricChecksum` $\mapsto_{string}$ `SearchLyric.SearchLyric.song` in `SearchLyricText`, `StrawBerry` executes a test for it by invoking `SearchLyric` passing as `song` the value of `LyricChecksum` obtained as result of `SearchLyricText` on an instance pool data. It gets an error answer and therefore it removes this dependency. After this step, all dependencies whose test case produced an error message are eliminated. When deleting the last dependency that participates in a connection between two nodes, also the arc between these two nodes must be removed. Nodes that have no incoming and outgoing arc can be removed. For the nodes, different from `start`, that have outgoing arcs and no incoming arc, except for loops, `StrawBerry` adds an incoming arc from `start` and adds the corresponding *certain* dependencies into `start`. Note that `start` can still contain *uncertain* dependencies.

Focusing on our example, all the dependencies in `start` that have `GetLyric` as sink operation are removed (as the corresponding arcs).

All the uncertain dependencies except for the ones stored in `start` are removed, and the corresponding arcs as well. A node is completely validated when it stores either only *certain* dependencies or no dependency.

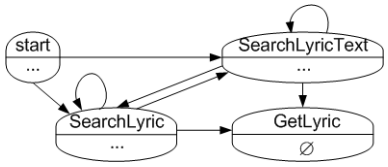

Figure 6: Validated dependencies automaton

Step 4.2, true dependencies confirmation: this step performs a first trivial check. `start` is marked as validated and all its dependencies become *certain*. Then, `StrawBerry` exercises every remaining uncertain dependency in every node through a negative test. For example, it executes a test for the dependency `SearchLyric.SearchLyricResponse_LyricId` $\mapsto_{int}$ `GetLyric.GetLyric_lyricId`. By providing as input to the `GetLyric` operation a randomly generated input of type `int`, `StrawBerry` gets an error answer and therefore it promotes to *certain* this dependency. After this step, all dependencies whose negative test case produced an error answer are confirmed as certain.

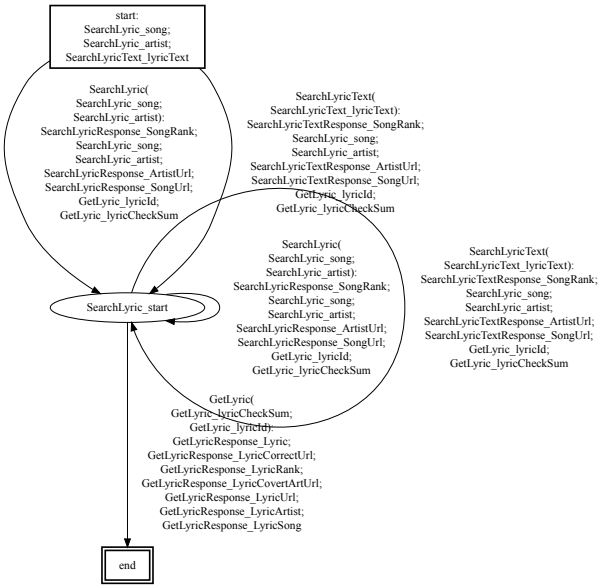

Figure 7: Behavior protocol automaton of `ChartLyrics`

Step 4.3, solving remaining uncertain dependencies: dependencies, if any, that remain uncertain after steps 4.1 and 4.2 refer to cases in which the testing of the sink operation of a dependency did not distinguish between the output produced by the source operation or a random input. In such (experimentally few) remaining cases, `StrawBerry` resolves the uncertainty by assuming that the hypothesized dependency does not exist.

Figure 6 shows the validated dependencies automaton obtained after the testing activity.

Activity 5: Behavior Protocol Synthesis.
This activity takes as input the validated dependencies automaton and automatically produces, as output, a corresponding control-flow model called *behavior protocol automaton*. It models

the behavior protocol of the WS, i.e., the interaction protocol that a client has to abide by to correctly interact with the WS. In Figure 7, we show this automaton for `ChartLyrics`. This automaton explicitly models also the data that has to be passed to the WS operations. Each arc label follows the syntax: `operation_name '(' semi-colon_separated_inputs ')' ':' semi-colon_separated_outputs`. The synthesis algorithm reflects the validated data dependencies in conjunction with the induced operation invocation dependencies. A description and a formalization of the algorithm is presented in [12].

In Figure 7, the state labeled with `start` and the doubled rounded state (`end`) are the initial and final states, respectively. The label of `start` indicates also those parameters, and the relative operations, that can be directly provided by a client of the WS. By using `StrawBerry` again, we automatically produced the behavior protocol automaton of `LyricWiki`. For the sake of space, we only show a subset of this protocol in Figure 8. The subset in the figure will be used as an example in the next two sections.

3.2 Using Behavior Protocols to Build Protocol Mapping Script

In this phase the behavior protocol automata produced by `StrawBerry` are translated into the *Zot* operational format in order to make them suitable for the generation of the mapping scripts. As mentioned in Section 2.3, the mapping script generation needs, in addition to a description of a service interface and behavior protocol, a compatibility relation defined respectively on the input and output data of the expected and the actual service to be provided. This relation maps each piece of data on an univocal label. Consider for instance the inputs of operation `getSong` of `LyricsWiki`. They are strings representing the id, the author, the title and a checksum for a song. Similar inputs are accepted by the operation `GetLyric` of `ChartLyrics`, even if they have been assigned different names. In order to complete our translation we map each of `GetLyric` inputs on an univocal label and we do the same for each of `getSong` inputs, so, for instance, both `id` (input of `getSong`) and `GetLyric_lyricId` (input of `GetLyric`), which represent the id of a song, are mapped on the label *lyricId*. This translation step is carried on exploiting ontology-based reasoning, which we do not describe into details here, but can be found in our previous work [15]. The compatibility relations for input and return parameters of the example expected and actual services are reported respectively in Table 1 and Table 2. As it can be noticed reading Table 2 these relations can be partial. Consider for instance `ArtistUrl`, on the third row of the table. The expected service has no correspondence for this piece of data, consequently, when this parameter is returned by an operation of the actual service, it should be discarded by the adapter.

At this point we are ready to identify the proper semantics, synchronous or asynchronous to be associated to transitions. As the Behavior Protocols generated by `StrawBerry` highlight the dependencies between input and output of operations, we can rely on their analysis to come up with the proper solution.

Consider for instance the operations `checkSongExist` and `getSong` in the protocol of `LyricsWiki`. The former returns two pieces of data (i.e., `id` and `checkcode`) which are also required as input by `getSong`. The semantics in this case is that the data returned by `checkSongExist` should be given as input to `getSong`. Consequently the latter should wait for `checkSongExist` to return before being invoked. This mandates that the asynchronous semantics does not fit this case, while a synchronous semantics is more appropriate.

`LyricsWiki` *parameter*	`ChartLyrics` *parameter*	*Univocal label*
song	song	song
checkcode	lyricCheckSum	lyricCheckSum
artist	artist	artist
searchString	lyricText	searchString
id	lyricId	lyricId

Table 1: Compatibility relation for input parameters of the expected and actual services

`LyricsWiki` *return parameter*	`ChartLyrics` *return parameter*	*Univocal label*
artist	artist	artist
song	song	song
-	ArtistUrl	ArtistUrl
-	LyricCovertArtUrl	LyricCovertArtUrl
id checkcode	lyricId lyricCheckSum	lyricId lyricCheckSum
LyricCorrectUrl	LyricUrl	LyricCorrectUrl
amazonLink	SongUrl	SongUrl
-	LyricRank	LyricRank
	ArtistUrl	ArtistUrl
-	SongRank	SongRank
lyrics	Lyric	Lyric
year	-	year
AlbumDataArray_item	-	item
album	-	album

Table 2: Compatibility relation for input parameters of the expected and actual services

3.3 Mapping Script Generation

The model described in previous section is analyzed to generate a mapping script that allows to substitute the expected service with the actual service, so that a client expecting to use the former can use the latter in its place. In order to be used in the place of an expected service, a actual service should support an operation sequence which accepts as input a subset of the input data provided by the client, and should return a superset of the data expected by the client. Given the behavior protocols of both the expected and actual service, the compatibility relations defined between data and states of the services and an expected operations sequence seq_{exp}, the approach formulates the problem of finding a substitutable actual operations sequence seq_{act}. If this sequence exists, a mapping script is generated.

Let us consider as an example the expected service operation sequence `checkSongExists`, `searchSongs`, `getSong`, which brings the `LyricsWiki` behavior protocol automaton from state `start` to state `checkSongExists_searchSongs_searchArtists_start` (see Figure 8). Moreover we will assume to have established a compatibility relation between services data, as shown in Table 1 and Table 2, and a state compatibility relation. The latter relation states that the `checkSongExists_searchSongs_searchArtists_start` state of the expected service is compatible with the `end` state of the actual service, which means that if the expected service reaches the `checkSongExists_searchSongs_searchArtists_start` state, then the actual service should reach the `end` state. The example expected operations sequence starts from the *start* state and leads the behavior protocol model into state `checkSongExists_searchSongs_searchArtists_start`. A client expecting to invoke this sequence is assuming to provide as input to the first operation of the sequence a song and an artist. Moreover it expects the invoked operation to return an `id` and a `checkcode`. Considering the actual service protocol, our approach searches for

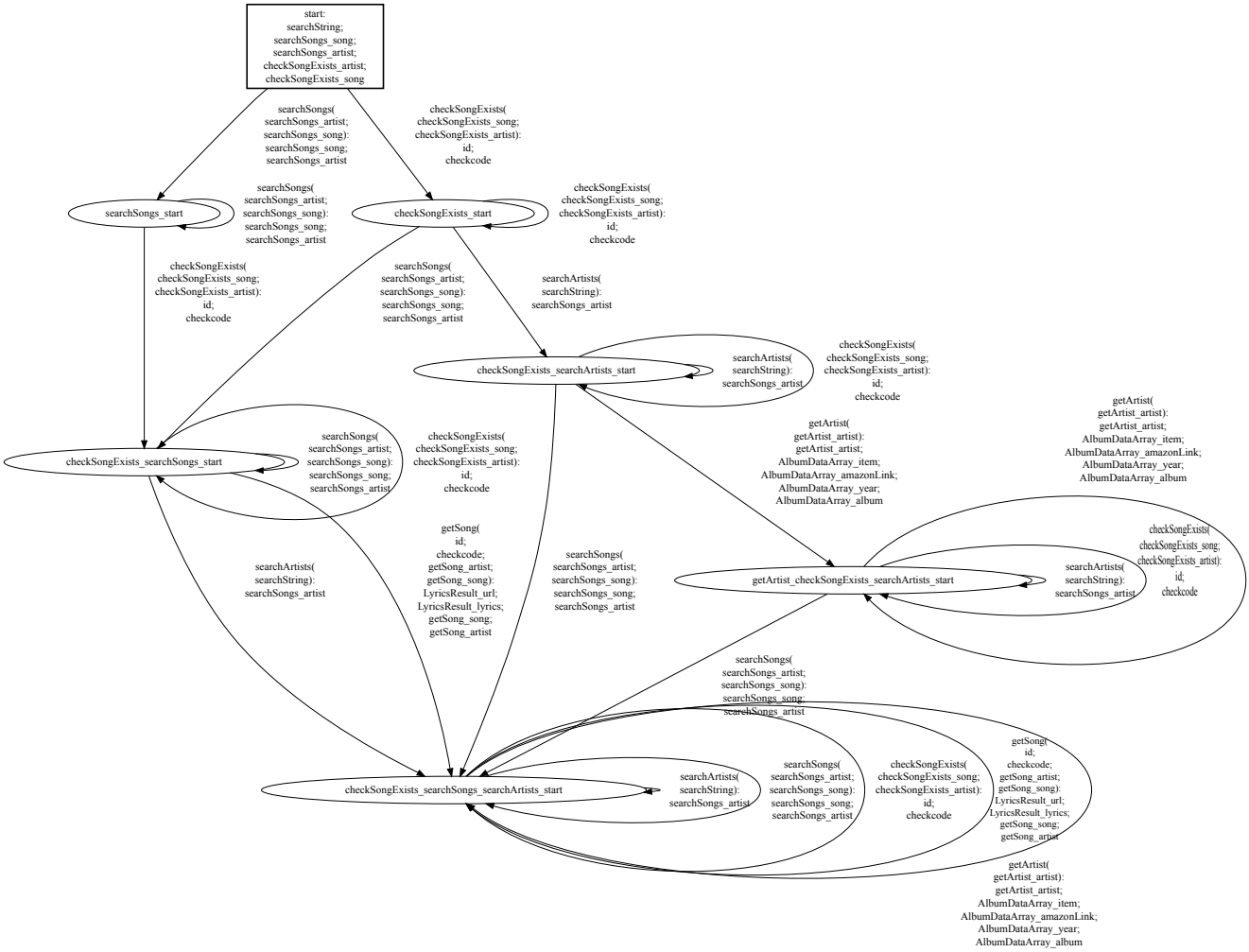

Figure 8: A subset of Behavior protocol automaton of `LyricWiki`

an operation accepting a subset of the provided input data and providing a superset of the required return data. The operation to be selected should leave the *start* state as the state compatibility relation provided as input for the approach mandates the compatibility of state *start* of `LyricsWiki` with state *start* of `ChartLyrics`. In our example the invocation of `checkSongExists` makes `SearchLyric` the only candidate for being selected. The latter returns also some piece of data that are not required by the invoked expected service operation. These piece of data are discarded and are not considered anymore in the adapter search procedure. After the invocation of `SearchLyric` the actual service goes in `SearchLyric_start` state.

The next operation on the expected sequence to be invoked is `searchSongs`, which requires as input the names of the song to be searched and of its author and provides as return parameters the name of the artist and of the song, if they are found. From the `SearchLyric_start` state of the actual service there are three operations that can be invoked: `SearchLyric`, `SearchLyricText` and `GetLyric`. In this situation `SearchLyric` is selected again, since it is the only one out of those three which complies to constraints on input and return parameters imposed by the invocation of `searchSongs` in the expected service. After the invocation of `SearchLyric` the actual service remains in the `SearchLyric_start` state.

The last operation in the expected sequence is `getSong`, which requires as input artist and song names and the id and checkSum returned by the previously invoked `checkSongExists`. The expected service has again the same three operations of the previous step available, but this time there are two available candidates for selection: `searchSongs` and `GetLyric`. In this situation the latter is selected, because of the state compatibility relation provided as input to the adapter search phase. Given the data-flow constraints elicited before, `GetLyric` is the only available operation that can satisfy also the state compatibility relation.

The search phase is performed using *Zot*. As stated before, *Zot* returns an execution trace which satisfies the given model, which is used to build the mapping script for the adapter (see Section 2.3 for details). A mapping script generated by *Zot* for the example sequence in this section is reported in Table 3. Each step contains the state in which each one of the analyzed automata (the ones of the expected and actual services) is, the operations in seq_{exp} and in seq_{act} that should be invoked in that step, and the exchanged data, if any. For each operation in seq_{exp} the adapter expects to receive an invocation for the expected service, for each operation in seq_{act} the adapter performs an invocation to the actual service.

Step	*Execution trace Content*
1	*LyricWiki*State:start ; *LyricWiki*Operation:checkSongExists *LyricWiki*Input: song, artist; *LyricWiki*Output:lyricId, lyricCheckSum *chartLyrics*State:start; *LyricWiki*Operation:checkSongExists
2	*LyricWiki*State:checkSongExists_start *chartLyrics*Input: song, artist *chartLyrics*Output:song , artist, artistUrl, songRank, lyricsId, lyricChecksum *chartLyrics*State:start; *chartLyrics*Operation:searchLyric
3	*LyricWiki*State:checkSongExists_start; *LyricWiki*Operation:searchSongs *LyricWiki*Input:song, artist; *LyricWiki*Output:song, artist *chartLyrics*State:searchLyric_start
4	*LyricWiki*State:checkSongExists_searchSongs_start *chartLyrics*Input:song, artist *chartLyrics*Output: song , artist, artistUrl, songRank, lyricsId, lyricChecksum *chartLyrics*State:searchLyric_start; *chartLyrics*Operation:searchLyric
5	*LyricWiki*State:checkSongExists_searchSongs_start; *LyricWiki*Operation: getSong *LyricWiki*Input: lyricId, song, lyricCheckSum, artist *LyricWiki*Output:song, artist, lyricCorrectUrl, Lyric *chartLyrics*State:searchLyric_start
6	*LyricWiki*State:checkSongExists_searchSongs_start *chartLyrics*Input: lyricId, lyricCheckSum *chartLyrics*Output: song , artist, artistUrl, lyricRank, Lyric, lyricCorrectUrl, lyricCoverArtUrl *chartLyrics*State:searchLyric_start *chartLyrics*Operation:getLyric
7	*LyricWiki*State:checkSongExists_searchSongs_start *chartLyrics*State:end

Table 3: Compatibility relation for input parameters of expected and actual services

4. RELATED WORK

Many approaches that support the automatic generation of adapters (or equivalent mechanisms) are based on the use of ontologies and focus on non-conversational services (see for instance our previous work [15] and [17]). They all assume that the usual WSDL definition of a service interface is enriched with some kinds of ontological annotations. At run-time, when a service bound to a composition needs to be substituted, a software agent generates a mapping by parsing such ontological annotations. *SCIROCO* [18] offers similar features but focuses on stateful services. It requires all services to be annotated with both a SAWSDL description and a WSResourceProperties [9] 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 as the services they consider are not conversational, they perform the mapping on a per-operation basis.

An approach that generates adapters covering the case of interaction protocols mismatches is presented in [13]. It assumes to start from a service composition and a service behavioral description both written in the BPEL language [8]. These are then translated in the *YAWL* formal language [26] 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 [21] offers similar features and has been implemented in an open source tool[6]. While both these approaches appear to fulfill our need for supporting interaction protocol mapping, they present some shortcoming in terms of both performances (as demonstrated in [14]) and constraints imposed on the nature of service specifications. On the contrary, thanks to the integration with `StrawBerry`, we are able to work with WSs specified only in terms of their WSDL, without making any assumption on the existence of additional specifications.

Concerning the problem of eliciting software behavioral models out of implemented systems, we briefly discuss here some of the works related to `StrawBerry`.

In [20], the authors describe a technique, called GK-Tail, to automatically generate behavioral models from system execution traces. GK-Tail assumes that execution traces are obtained by monitoring the system through message logging frameworks. Since the set of monitored traces represents only positive samples of the system execution, their approach cannot guarantee the complete correctness of the method. Instead the set of data dependencies, inferred by `StrawBerry`, concerns both positive and negative samples and it is syntactically correct by construction. However, it might not be correct semantically since it may contain false positives. These false positives are detected by the testing phase. Furthermore, dealing with black-box WSs, we cannot assume to take as input a set of interaction traces.

The work described in [19] (i.e., the SPY approach) aims to infer a formal specification of stateful black-box components that behave as data abstractions (Java classes that behave as data containers) by observing their run-time behavior. SPY is based on two assumptions: (i) the value of method parameters does not impact the implementation logic of the methods of a class; (ii) class instances have a kind of "uniform" behavior. In our context, we cannot rely on the previously mentioned assumptions.

The approaches described in [28, 29] analyze Java code to infer sequences of method calls. These sequences are then used to produce object usage patterns that serve to detect object usage violations in the code. Differently from `StrawBerry`, they are white-box methods.

The authors of [11] present an approach for inferring state machines with an infinite state space. The main difference between our work and this work is that we have the opposite problem of relaxing some dependence (when its existence is not certain) rather than adding new dependencies. The authors of [22] describe a learning-based black-box testing approach in which the problem of testing

[6]The Dinapter tool: http://sourceforge.net/projects/dinapter

functional correctness is reduced to a constraint solving problem. The testing phase of our approach shares some ideas with the approach described in [22], however we do not use *function approximation theory* and our coverage criterion is established by looking at the inferred I/O dependencies.

5. CONCLUSION AND FUTURE WORK

This paper presents a tool-supported methodology that takes as input the WSDL of a pair of conversational services and automatically builds a mapping script. This realizes a mapping between a sequence of operation invocations on a WS to be replaced and an equivalent sequence of operation invocations on the replacing WS. We have started to show, through its application to real web-services, that the methodology is viable: the behavior protocol elicitation nicely converges to a realistic automaton, and the mapping script generation is able to handle complex cases featuring complex protocols. The behavior protocol automaton elicitation for both `ChartLyrics` and `LyricWiki` required the execution of each test case, and each of them took a time in the order of 10^{-2} sec., while the total number of test cases was in the order of 10^3, summing up to few minutes of testing. Depending on the type of application, this can be perfectly acceptable for dynamic analysis. The mapping script generation for the example services took a time in the order of 1 sec, which makes it suitable for on-line use. We obviously need to carry out more empirical investigations to convey such preliminary evidences into a real quantitative assessment.

For future work we plan to investigate the introduction of heuristics for optimization in the protocol elicitation phase, as we believe that it is the first direction to push further to reduce the testing effort in the subsequent steps. Further, since services are usually invoked in complex processes that may feature a state or transactional support, service substitution may require house keeping work of the running processes. Thus, we plan to extend the mapping script generation process to allow consistent substitution of stateful and transactional services. Finally, we have not performed a rigorous evaluation of the approach yet. We simply base on an empirical evidence that the approach is feasible and, in some cases, it can give promising results. Thus, as future work, we consider also an extensive validation of the approach in order to establish, e.g., how accurate is the mapping between two WSs, how many scenarios that work on a WS do not work on the replacing WS, whether all dependencies marked as certain/uncertain are correctly classified, and the like.

6. REFERENCES

[1] eviware soapUI: http://www.soapui.org.

[2] The ChartLyric API. http://www.chartlyrics.com/api.aspx.

[3] The ChartLyric SOAP interface. http://api.chartlyrics.com/apiv1.asmx?WSDL.

[4] The ChartLyric song search engine. http://www.chartlyrics.com/.

[5] The Dinapter tool: http://sourceforge.net/projects/dinapter.

[6] The LyricWiki SOAP interface. http://lyrics.wikia.com/server.php?wsdl.

[7] The LyricWiki song search engine. http://lyrics.wikia.com/Main_Page.

[8] WS-BPEL specification. `http://www.oasis-open.org/committees/tc\_home.php?wg\_abbrev=wsbpel`.

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

[10] C. Bartolini, A. Bertolino, E. Marchetti, and A. Polini. WS-TAXI: a WSDL-based testing tool for Web Services. In *ICST 2009, Denver, Colorado - USA*. IEEE, 2009.

[11] T. Berg, B. Jonsson, and H. Raffelt. Regular Inference for State Machines Using Domains with Equality Tests. In *FASE 2008, Budapest, Hungary*, pages 317–331, 2008.

[12] A. Bertolino, P. Inverardi, P. Pelliccione, and M. Tivoli. Automatic synthesis of behavior protocols for composable web-services. In H. van Vliet and V. Issarny, editors, *ESEC/SIGSOFT FSE*, pages 141–150. ACM, 2009.

[13] A. Brogi and R. Popescu. Automated generation of BPEL adapters. In *In Proceedings of ICSOC*, 2006.

[14] L. Cavallaro, E. D. Nitto, and M. Pradella. An automatic approach to enable replacement of conversational services. In L. Baresi, C.-H. Chi, and J. Suzuki, editors, *ICSOC/ServiceWave*, volume 5900 of *Lecture Notes in Computer Science*, pages 159–174, 2009.

[15] L. Cavallaro, G. Ripa, and M. Zuccalà. Adapting service requests to actual service interfaces through semantic annotations. In *In Proceedings of PESOS*, 2009.

[16] V. De Antonellis, M. Melchiori, L. De Santis, M. Mecella, E. Mussi, B. Pernici, and P. Plebani. A layered architecture for flexible web service invocation. *Softw. Pract. Exper.*, 36(2):191–223, 2006.

[17] C. Drumm. *Improving Schema Mapping by Exploiting Domain Knowledge*. PhD thesis, Universitat Karlsruhe, Fakultat fur Informatik, 2008.

[18] M. Fredj, N. Georgantas, V. Issarny, and A. Zarras. Dynamic service substitution in service-oriented architectures. In *In Proceedings of SERVICES*, 2008.

[19] C. Ghezzi, A. Mocci, and M. Monga. Synthesizing Intentional Behavior Models by Graph Transformation. In *ICSE 2009, Vancouver, Canada*, 2009.

[20] D. Lorenzoli, L. Mariani, and M. Pezzè. Automatic Generation of Software Behavioral Models. In *ICSE 2008*, pages 501–510, NY, USA, 2008. ACM.

[21] J. A. Martìn and E. Pimentel. Automatic generation of adaptation contracts. In *Proceedings of FOCLASA*, 2008.

[22] K. Meinke. Automated Black-box Testing of Functional Correctness using Function Approximation. *SIGSOFT Softw. Eng. Notes*, 29(4):143–153, 2004.

[23] M. Pradella, A. Morzenti, and P. S. Pietro. The symmetry of the past and of the future: bi-infinite time in the verification of temporal properties. In *ESEC/SIGSOFT FSE*, pages 312–320, 2007.

[24] M. Pradella, A. Morzenti, and P. S. Pietro. Refining real-time system specifications through bounded model- and satisfiability-checking. In *ASE*, pages 119–127, 2008.

[25] M. Utting and B. Legeard. *Practical Model-Based Testing - A Tools Approach*. Morgan and Kaufmann, 2006.

[26] W. M. P. van der Aalst and A. H. M. ter Hofstede. Yawl: yet another workflow language. *Information Systems*, 30(4):245–275, 2005.

[27] K. Verma, K. Gomadam, A. Sheth, J. Miller, and Z. Wu. The meteor-s approach for configuring and executing dynamic web processes. Technical report, LSDIS Lab, University of Georgia, Athens, Georgia, 2005.

[28] A. Wasylkowski and A. Zeller. Mining Operational Preconditions. http://www.st.cs.uni-saarland.de/models/papers/wasylkowski-2008-preconditions.pdf (Tech. Rep.).

[29] A. Wasylkowski, A. Zeller, and C. Lindig. Detecting Object Usage Anomalies. In *ESEC-FSE '07, pp. 35-44*. ACM, 2007.