Automatic Synthesis of Behavior Protocols for Composable Web-Services

Antonia Bertolino, Paola Inverardi, Pelliccione Patrizio, Tivoli Massimo

Automatic Synthesis of Behavior Protocols for Composable Web-Services

Antonia Bertolino[1], Paola Inverardi[2], Patrizio Pelliccione[2], Massimo Tivoli[2]
[1] Istituto di Scienza e Tecnologie dell'Informazione "A. Faedo"
Consiglio Nazionale delle Ricerche - Via Moruzzi 1 - 56124 Pisa, Italy
antonia.bertolino@isti.cnr.it
[2] Università dell'Aquila, Dipartimento di Informatica
Via Vetoio, L'Aquila, Italy
{paola.inverardi, patrizio.pelliccione, massimo.tivoli}@di.univaq.it

ABSTRACT

Web-services are broadly considered as an effective means to achieve interoperability between heterogeneous parties of a business process and offer an open platform for developing new composite web-services out of existing ones. In the literature many approaches have been proposed with the aim to automatically compose web-services. All of them assume that, along with the web-service signature, some information is provided about how clients interacting with the web-service should behave when invoking it. We call this piece of information the web-service *behavior protocol*. Unfortunately, in the practice this assumption turns out to be unfounded. To address this need, in this paper we propose a method to automatically derive from the web-service signature an automaton modeling its behavior protocol. The method, called `StrawBerry`, combines synthesis and testing techniques. In particular, synthesis is based on data type analysis. The conformance between the synthesized automaton and the implementation of the corresponding web-service is checked by means of testing. The application of `StrawBerry` to the *Amazon E-Commerce Service* shows that it is practical and realistic.

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.

Keywords

Web-services, Automatic synthesis, Testing, Behavior protocols.

ESEC-FSE'09, August 23–28, 2009, Amsterdam, The Netherlands.

1. INTRODUCTION

The near future in software production envisions an ubiquitous world of available services that can be easily discovered and assembled to fit one user's needs. The Internet of Services paradigm [2] emerges from the convergence of the future Internet and of the service-oriented software paradigm. Web-services (WSs) play a central role in this vision as effective means to achieve interoperability between heterogeneous parties of a business process and independence from the underlying infrastructure. At the same time they offer an open platform for composing new value added WSs that can enhance the amount and quality of the available WSs.

A key challenge towards this future direction concerns the possibility to characterize WSs semantically. This means that besides a syntactical description of the WS signature, e.g., by means of the WSDL notation [3], there is the need for pieces of semantic information about the functionality the WS provides. These pieces of information can describe different views of the WS semantics, from ontological ones that ease the discovery phase, to behavioral ones that ease the composition of WSs. For the latter many approaches have been proposed in the literature with the aim to automatically compose WSs, see [6, 7, 10, 12, 13] just to mention the most recent. In most of the cases, the approaches rely on the assumption that, along with the WSDL, some information is provided about how the clients invoking the WS should behave. We call this behavioral information the WS *behavior protocol*. Unfortunately, in the software practice this assumption turns out to be unfounded.

This is the problem we address here: how to compensate for the lack of information about a WS's behavioral assumptions. In this paper we present a method, `StrawBerry` (Synthesized Tested Refined Automaton of Web-service BEhavior pRotocol), for the automatic discovery of the behavior protocol of a WS. Since for a published WS, in practice, only its WSDL description can generally be assumed to be available, `StrawBerry` derives from the WSDL, 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 abide by to correctly interact with the WS. This automaton explicitly models also the data 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 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 focusses 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 [9, 15, 16]. In fact, the behavior protocol obtained through `StrawBerry` enables the automatic composition of WSs. Indeed, as Business Process Execution Language (BPEL) [1] orchestration is today the emerging standard approach to composite WS development, our eventual goal, that is beyond the scope of the present paper, is to use the synthesized behavior protocols for automatic BPEL orchestration purposes. Therefore `StrawBerry` has been conceived to support the design-time activities of the BPEL process architect.

The paper is organized as follows. Section 2 presents the actual technological scenario in which `StrawBerry` works and discusses the underlying programming assumptions. In Section 3, by means of an explanatory example, we introduce the `StrawBerry` method. Section 4 presents the method formalization. In Section 5, we show an application of `StrawBerry` to an existing WS, i.e., the *Amazon E-Commerce Service*. In Section 6, we relate `StrawBerry` to other similar approaches. In Section 7, we conclude and outline future research directions.

2. SETTING THE CONTEXT

A WS is typically constructed over HTTP and it is by default a *state-less* software entity. That is, no state internal to the WS exists during a *complex* interaction with the same client. This is not the best solution for application scenarios such as e-commerce. Some technologies have been proposed to allow the development of *state-full* WSs through the implementation of the concept of *session*. Informally, a session consists of a set of attributes (set of data with name, type, and value) that characterize an invoked sequence of WS operations. Typically, a session is realized to be persistent during a complete WS interaction with the client. Different approaches have been proposed to realize a session: (i) by using the well-known mechanism of *cookies*; (ii) by using WS-oriented APIs[1], or, at a lower level, by means of session IDs associated to the header of the SOAP messages; (iii) based on the WS-ReliableMessaging standard[2]; or (iv) by ad-hoc programming, that mixes data used for managing the session with business logic data[3].

Each of these approaches has its own advantages and disadvantages. Techniques (i) and (ii) keep the business logic separated from the logic used to manage the session. However, in order to use these techniques, the client code must be aware of the session WS capabilities. Technique (iii) also keeps the business logic separated from the session management logic. Furthermore, session management is completely transparent to the client since it is demanded to a framework on top of which the WS is built. However the client application must support the particular implementation of the framework. Using technique (iv), WSs maintain their *state-less* nature, and a session is implicitly realized by passing the relative data (i.e., data encoding the WS state) from one operation to another. Therefore, session data are explicitly added as I/O data of the WSDL operations. The disadvantage of this technique is that the data concerning both business and session logic are mixed in the WSDL. However the client application needs only to rely on the WSDL interface in order to interact with the WS. This promotes WS reuse and interoperability among different services.

It is worthwhile noticing that BPEL orchestration cannot be realized with WSs developed by using techniques (i), (ii), and (iii). A BPEL process cannot use such (hidden) techniques to enable a session management. Since the present standard for WS composition is BPEL and an important number of relevant WSs, like Amazon, follows technique (iv), this is also the programming assumption for `StrawBerry`.

[1]e.g., JAX-WS: http://weblogs.java.net/blog/ramapulavarthi/archive/2006/06/maintaining_ses.html

[2]WS-ReliableMessaging standard: http://weblogs.java.net/blog/mikeg/archive/2006/08/wsreliable_mess.html

[3]As it is done for the *Amazon* e-commerce service: http://webservices.amazon.com/AWSECommerceService/AWSECommerceService.wsdl.

3. METHOD DESCRIPTION

In this section we provide an overview of the `StrawBerry` method (Sect. 3.1). Then, by means of a simple explanatory example (Sect. 3.2), we informally introduce its steps (Sect. 3.3).

3.1 Overview

`StrawBerry` takes as input the WSDL of a WS, and returns an automaton modeling its behavior protocol (client side). Figure 1 graphically represents `StrawBerry` as a process split in five main activities. 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 be optimized under some heuristics concerning the syntactic characteristics of the WSDL. The elicited set of I/O dependencies (see the *Input/Output Dependencies* artifact shown in Fig. 1) is used for constructing a data-flow model (see the *Saturated Dependencies Automaton Synthesis* activity and the *Saturated Dependencies Automaton* artifact shown in Fig. 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 providing directly 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 Fig. 1).

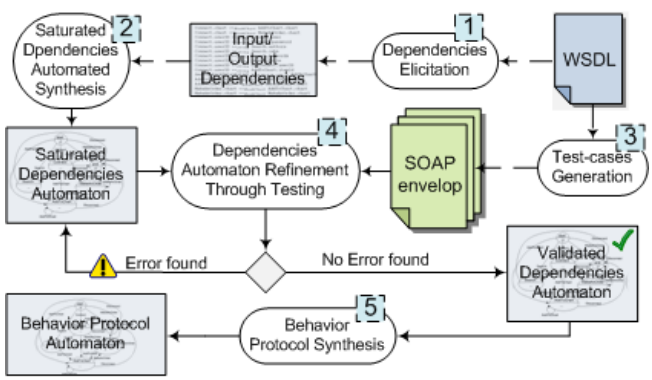

Figure 1: Overview of the method

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 [4] 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 [14]. 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, the oracle is based on the implementation and testing aims at validating whether the synthesized automaton is a correct data-flow abstraction of it. 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. 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 Fig. 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.

3.2 Explanatory example

The explanatory example that we use in this paper is a WS of an online bookshop that we call WS_Lib. This WS defines, in its WSDL, the following operations:

• Connect: in order to login, a registered user inserts its/her username and password. This operation returns an empty cart and a userID (to uniquely identify the session). If something wrong occurs (e.g., wrong user or password), an error message is returned; an output parameter of name errmsg is included also for all the other operations.

Input data	Output data
user: string;	cart: BookCart;
password: string;	userID: string;
	errmsg: string;

• Disconnect: it is used by a logged user to logout and close the session.

Input data	Output data
userID: string;	regularResponse: string;
	errmsg: string;

• Search: it is used to search the bookshop catalogue by means of different search criteria, namely, by authors, ISBN, keywords, and title, and returns a list of books satisfying the search criteria.

Input data	Output data
authors: string;	bookDetailsList: BookDetailsList;
isbn: string;	errmsg: string;
keywords: string;	
title: string;	

• AddToCart: it adds a book from a list to the cart associated to the user.

Input data	Output data
itemId: string;	cart: BookCart;
itemList: BookDetailsList;	errmsg: string;
cart: BookCart;	

• MakeAnOrder: it makes an order of the books contained in the cart. When the order has been made the cart is emptied.

Input data	Output data
cart: BookCart;	cart: BookCart;
	errmsg: string;

Data that characterize a WS session are: userID (user identifier), cart (the cart associated to the user), bookDetailsList and itemList (the list of books that match the search criteria).

3.3 Stepwise description

By referring to Fig. 1, we show an overview of how our approach would process the WS_Lib WSDL.

Activity 1: Dependencies Elicitation.
This activity is split in the following 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.

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. Starting from the flattened WSDL, by syntactically matching the type of an output element of an operation with the type of an input element of another operation, StrawBerry automatically elicits the following set of data dependencies:

Connect.cart $\mapsto_{BookCart}$ AddToCart.cart
Connect.cart $\mapsto_{BookCart}$ MakeAnOrder.cart
Connect.userID $\mapsto_{string}$ x, for each x$\in I_{string}$
Connect.errmsg $\mapsto_{string}$ x, for each x$\in I_{string}$
Disconnect.regularResponse $\mapsto_{string}$ x, for each x$\in I_{string}$
Disconnect.errmsg $\mapsto_{string}$ x, for each x$\in I_{string}$
Search.bookDetailsList $\mapsto_{BookDetailsList}$ AddToCart.itemList
Search.errmsg $\mapsto_{string}$ x, for each x$\in I_{string}$
AddToCart.cart $\mapsto_{BookCart}$ AddToCart.cart
AddToCart.cart $\mapsto_{BookCart}$ MakeAnOrder.cart
AddToCart.errmsg $\mapsto_{string}$ x, for each x$\in I_{string}$
MakeAnOrder.cart $\mapsto_{BookCart}$ AddToCart.cart
MakeAnOrder.cart $\mapsto_{BookCart}$ MakeAnOrder.cart
MakeAnOrder.errmsg $\mapsto_{string}$ x, for each x$\in I_{string}$

where: I_{string}={Connect.user,Connect.password,Search.authors, Search.isbn,Search.keywords,Search.title,Disconnect.userID, AddToCart.itemId}.

For instance, Search.bookDetailsList $\mapsto_{BookDetailsList}$ AddToCart.itemList means that, the value of bookDetailsList, as output of Search, can be set as input parameter itemList of AddToCart.

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

Step 1.2, dependency set optimization: this optimization step can be enabled/disabled by the StrawBerry's user. It aims at identifying those dependencies that can be already removed or considered as certain, hence preventing waste useless test resources in the fourth activity of our method. This step is currently based on the following three heuristics (as we accumulate further experience, smarter heuristics can be introduced).

• *Heuristic 1*: all dependencies defined on a "complex type" are considered as certain. This heuristic comes out from the observation that a dependency defined on a complex type is certain with a high probability due to the specificity of that type. In our approach, as complex type, we consider XML Schema types defined by means of the complexType and simpleType tags (e.g., sequence, choice, all, restriction and extension of a base type).

• *Heuristic 2*: all dependencies defined between data parameters having the same name (and the same type) are considered as certain. This heuristic comes out from usual programming practice.
• *Heuristic 3*: all dependencies defined between an output parameter that is interpreted as an error and an input parameter can be removed. In our example, `errmsg` is a string. Error outputs should be never matched since they represent the end of an interaction. Thus, if the `StrawBerry`'s user has this information this heuristic can be enabled to prune the set of dependencies.

Coming back to our explanatory example, after the application of *Heuristic 1*, the following dependencies are considered as certain:

$\texttt{Connect.cart} \mapsto_{BookCart} \texttt{AddToCart.cart}$
$\texttt{Connect.cart} \mapsto_{BookCart} \texttt{MakeAnOrder.cart}$
$\texttt{Search.bookDetailsList} \mapsto_{BookDetailsList} \texttt{AddToCart.itemList}$
$\texttt{AddToCart.cart} \mapsto_{BookCart} \texttt{AddToCart.cart}$
$\texttt{AddToCart.cart} \mapsto_{BookCart} \texttt{MakeAnOrder.cart}$
$\texttt{MakeAnOrder.cart} \mapsto_{BookCart} \texttt{AddToCart.cart}$
$\texttt{MakeAnOrder.cart} \mapsto_{BookCart} \texttt{MakeAnOrder.cart}$

Note that in this case it is possible to perform a `MakeAnOrder` operation after another `MakeAnOrder` even though the `cart` is empty (the same hold for a `MakeAnOrder` operation after `Connect`) since the WS implementation considers this behavior as correct. In another scenario this could be considered as an error thus preventing the use of *Heuristic 1*.

Now, if we apply *Heuristic 2*, another dependency is considered as certain: $\texttt{Connect.userID} \mapsto_{string} \texttt{Disconnect.userID}$. Finally, if we apply *Heuristic 3*, the following dependencies are removed:

$\texttt{Connect.errmsg} \mapsto_{string} \texttt{x}$, for each $\texttt{x} \in I_{string}$
$\texttt{Disconnect.errmsg} \mapsto_{string} \texttt{x}$, for each $\texttt{x} \in I_{string}$
$\texttt{Search.errmsg} \mapsto_{string} \texttt{x}$, for each $\texttt{x} \in I_{string}$
$\texttt{AddToCart.errmsg} \mapsto_{string} \texttt{x}$, for each $\texttt{x} \in I_{string}$
$\texttt{MakeAnOrder.errmsg} \mapsto_{string} \texttt{x}$, for each $\texttt{x} \in I_{string}$

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 Fig. 2, we show the nodes built for `WS_Lib`. A node stores the name of the operation and the data dependencies defined on its output parameters. In Fig. 2, *certain* dependencies are identified by the tick (✓).

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 `WS_Lib` is shown in Fig. 3. The `Env` node and the dotted lines represent the node and the arcs added by the saturation phase explained in the following step.

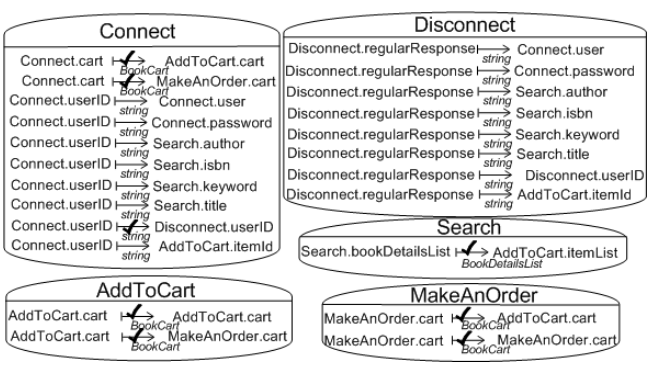

Figure 2: Generated Nodes

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, `Env`. 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 `Env`. According to the dependencies stored into `Env`, the saturation step adds an arc from `Env` to every other depending node.

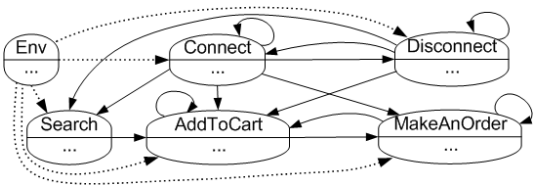

Figure 3: Saturated dependencies automaton

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 envelop 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[4] is probably the most popular. `StrawBerry` adopts the WS-TAXI tool [4] 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 [4]. It is worth however to clarify how WS-TAXI deals with input parameter values.

Listing 1 shows an example of a SOAP envelop message produced by WS-TAXI for testing `Search`. This test case aims at performing a book search based on the authors criterion. In Listing 1, the authors parameter is 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 AddToCart SOAP envelop message

```
<soapenv:Envelope
xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:lib="http://www.example.org/Lib/">
  <soapenv:Header/>
  <soapenv:Body xmlns="http://www.example.org/Lib/">
    <SearchRequest>
      <authors>KOVjotMZBEfbeynkhtAviBIEs</authors>
    </SearchRequest>
  </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* [8], is available. For example, in the case of `Connect`, it is necessary to use a correct pair of `user` and `password`. Typically, the credentials to access a WS are provided when making the registration for using it, as done for the *Amazon e-commerce service*. Thus, we assume to have an instance pool of valid `user/password` pairs. Other instance pools of different nature can be reasonably provided by an application user or a

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

domain expert. For instance, it is easy to produce a list of books probably contained into Amazon. Wrapping up, if an instance pool is available for some operations, StrawBerry exploits this useful piece of information feeding the WS-TAXI database. For the WS_Lib example, we provide the instance pool for Connect and Search, as shown in Table 1. Back to Listing 1, the authors parameter can be now taken directly from the instance pool in Table 1, thus producing more realistic test cases.

Operation	Input Data	Operation	Input Data
Connect	u: Antonella; p: anto u: Massimo; p: Max u: Paola; p: paolina u: Patrizio; p: P@ ...	Search	auth: Jean-Paul Sartre auth: R. Sennett, J. Cobb auth: Noam Chomsky auth: J. David Salinger ...

Table 1: 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, as explained below. Note that since we start from the saturated automaton and 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 cart) and negative responses to well-formed requests (e.g., a search of a book 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 this paper we propose a partial, but general, oracle that is based on the following observations: (i) whenever invoking different operations with wrong input data, the error answer message is (almost) always the same; (ii) error answers are typically short; (iii) regular answers are typically different from each other; (iv) regular answers are typically long. This partial oracle can be realized by using a statistical approach to partition WS responses into regular and error answers. In this paper, we do not discuss the actual implementation of such a general oracle.

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. Alternatively, we could investigate further by launching more test cases and making a statistic inference from the observed results. An empirical evaluation of the impact of this third step, and a possible improvement of StrawBerry with a significance test for the uncertain dependencies that reach the third step, is left to future work.

Step 4.1, false dependencies elimination: each *uncertain* dependency in every node is tested. For example, considering the dependency Connect.userID $\mapsto_{string}$ Search.isbn in Connect, StrawBerry executes a test for it by invoking Search passing as isbn the value of userID obtained as result of Connect 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 Env, that have outgoing arcs and no incoming arc, except for loops, StrawBerry adds an incoming arc from Env and adds the corresponding *certain* dependencies into Env. Note that Env can still contain *uncertain* dependencies.

Focusing on our example, all the dependencies in Env that have MakeAnOrder and Disconnect as sink operations are removed (as the corresponding arcs). Thus, the survived dependencies into Env are:

★$\mapsto_{string}$AddToCart.itemId,
★$\mapsto_{string}$Search.p, p∈{authors,isbn,keywords,title},
★$\mapsto_{string}$Connect.p, p∈{user,password}.

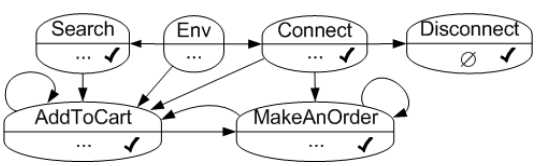

Figure 4: Dependencies automaton after step 4.1

All the uncertain dependencies except for the ones stored in Env are removed. Thus, some arcs shown in Fig. 3 are removed leading to the automaton shown in Fig. 4. A node is validated when it stores either only *certain* dependencies or no dependency. After this step, the only non-validated node is Env as shown in Fig. 4 where validated nodes are marked with ✓. Validation in this step is essentially due to the good functioning of the heuristics.

Step 4.2, true dependencies confirmation: this step performs a first trivial check. Env is marked as validated and all its dependencies become *certain*. By considering the automaton shown in Fig. 4, this means that we can conclude the testing activity. However, if we had applied StrawBerry without heuristics, we would have had for instance that Connect.userID $\mapsto_{string}$ Disconnect.userID (which has been promoted, in step 4.1, as *certain* by Heuristic 2) could not be deleted since the test did not fail, and therefore Connect would have not been validated. In this case, StrawBerry exercises every remaining uncertain dependency in every node through a negative test. For example, it executes a test for the dependency Connect.userID $\mapsto_{string}$ Disconnect.userID. By providing as input to the Disconnect operation a randomly generated input of type String, 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.

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.

Activity 5: Behavior Protocol Synthesis.
This activity takes as input the validated dependencies automaton. For each operation *op* in the automaton, this activity takes into account the operations that are required to produce the input parameters of *op*. For instance, for `AddToCart`, the validated dependencies where `AddToCart` is a sink operation are:
$\bigstar \mapsto_{string}$`AddToCart.itemId`,
`Connect.cart` $\mapsto_{BookCart}$ `AddToCart.cart`,
`AddToCart.cart` $\mapsto_{BookCart}$ `AddToCart.cart`,
`Search.bookDetailsList` $\mapsto_{BookDetailsList}$ `AddToCart.itemList`.
By looking at these dependencies, this activity elicits that, in order to invoke `AddToCart`, `itemId` must be provided by the client, `cart` can be set by the output `cart` of either `Connect`, `AddToCart` itself, or `MakeAnOrder`, and `itemList` is set by the output `bookDetailList` of `Search`. In Fig. 5, we graphically represent the operations that should be invoked before invoking `AddToCart` (see table $T_{AddToCart}$) according to the dependencies validated on its input parameters `itemId`, `itemList`, and `cart`. An analogous process is performed for the other operations hence leading to produce the information graphically represented in Fig. 5.

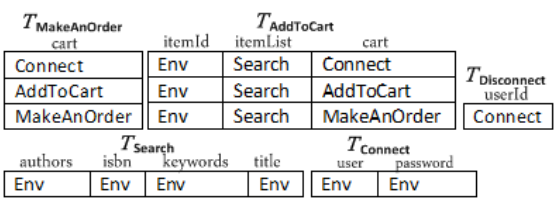

Figure 5: Operation invocation dependencies

This information is used to synthesize an automaton that 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 Fig. 6, we show this automaton for our explanatory example. This automaton explicitly models also the data that has to be passed to the WS operations. Each arc label follows the syntax: `operation_name '(' comma_separated_inputs ')' ':' comma_separated_outputs`. The synthesis algorithm reflects the validated data dependencies in conjunction with the operation invocation dependencies represented in Fig. 5. The algorithm is presented in Sect. 4. For the sake of readability, in Fig. 6, we omit I/O data for some operation and in place of a data parameter name we use its initials.

In Fig. 6, the state with the (no-source) incoming arrow and the doubled circled state are the initial and final states, respectively. Note that, in general, the WSDL of the WS can define operations that are not taken into account by the validated dependencies automaton since these operations are not involved in any dependency because they can be always invoked. In order not to complicate a behavior protocol automaton, this aspect is reflected by implicitly considering that these operations become loop transitions on every state of the automaton.

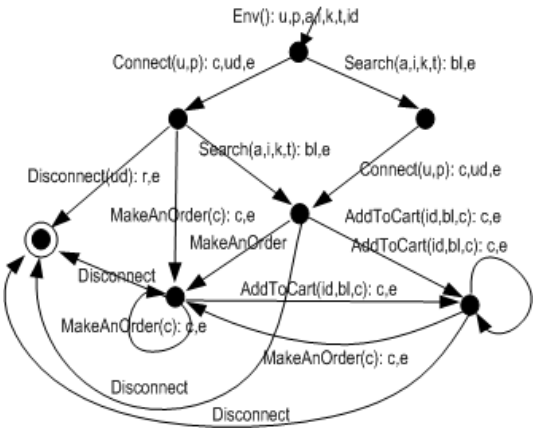

Figure 6: Behavior protocol automaton

4. METHOD FORMALIZATION

In this section we formalize the `StrawBerry` method. This formalization rigorously defines all the method stages concluding with a detailed presentation of the `StrawBerry` testing process and of the behavior protocol automaton synthesis. Furthermore, it represents the specification from which the prototypal implementation of `StrawBerry` has been realized. For the sake of simplicity, we omit the formalization of the three heuristics discussed in Sect. 3.3 since it is straightforward.

Let W be a WSDL interface, we denote with $\mathrm{Op_W}$ the set of all the operation names of W.

We denote with $\mathfrak{D}_W$ the set of all I/O data dependencies of W obtained by syntactically matching the type of an output parameter of an operation in $\mathrm{Op_W}$ with the type of an input parameter of another operation in $\mathrm{Op_W}$. $\mathfrak{D}_W$ can be partitioned into C_W and U_W that denote the set of all the *certain* and *uncertain* dependencies, respectively. Thus, with either $op.p \not\mapsto_t op'.p'$ or $\bigstar \not\mapsto_{t'} op'.p'$ we denote elements of C_W for some $op,op' \in Op_W$ and parameter names p,p' of type t. Analogously, with either $op.p \mapsto_t op'.p'$ or $\bigstar \mapsto_{t'} op'.p'$ we denote elements of U_W. Note that this notation implies that p is the name of an output parameter of op and p' is the name of an input parameter of op'.

Hereafter if $op.p \mapsto_t op'.p'$ ($op.p \not\mapsto_t op'.p'$), we write that "**a dependency exists**" for op. We can also write that op' "**depends on**" op. If $\bigstar \mapsto_{t'} op'.p'$ ($\bigstar \not\mapsto_{t'} op'.p'$), we write that op' "**is dependent**" on the environment.

Once the set of I/O dependencies has been built we can construct the *Dependencies Automaton* as defined in Def. 2. This definition makes use of the function *Node generator* defined in Def. 1. The role of this function is to define the nodes of the automaton that will be built by Def. 2. A special case is the node `Env` that is directly added by a saturation rule, see Def. 3. As described in Sect. 3 *Node generator* implements the *Step 2.1, node generation* of the *Activity 2*. The operations for which a node must be built are identified by means of the *I/O Dependency set* $\mathfrak{D}_W$.

DEFINITION 1 (NODE GENERATOR).
Node generator N_{gen}:$\mathrm{Op_W} \longrightarrow 2^{\mathfrak{D}_W}$, *is a function that given as input* $op \in Op_W$ *returns the set* $\mathrm{D} \in 2^{\mathfrak{D}_W}$ *s.t. either* D *is empty or for each* $\mathrm{dep} \in \mathrm{D}$, *a* dependency exists *for op and it does not exist* $\mathrm{D}' \in 2^{\mathfrak{D}_W}$ *s.t. for each* $\mathrm{dep}' \in \mathrm{D}' \setminus \mathrm{D}$, *a* dependency exists *for op.*

At the beginning, all the dependencies stored into a node generated by *Node generator* are *uncertain*. `StrawBerry` makes use of the previously discussed three heuristics in order to set to *certain* some dependencies and to remove some others. We recall that by

exploiting the *Node generator* function and the *I/O Dependency set*, `StrawBerry` synthesizes an automaton that models all the chains of data dependencies that should be taken into account while using the WS. Each arc from a node to another node reflects the I/O Dependency set, thus we call this automaton the *Dependencies automaton*.

DEFINITION 2 (DEPENDENCIES AUTOMATON).
A Dependencies automaton $A_W=(N,\Delta)$ *of a WSDL interface* W *is an automaton where:*
► $N=\{(op_1,N_{gen}(op_1)),\cdots,(op_v,N_{gen}(op_v))\}$ *is the set of nodes s.t.* $\{op_1,\cdots,op_v\}\subseteq \mathrm{Op_W}$ *and for each* $i=1,\cdots,v$ *either a* dependency exists *for* op_i *or* op_i depends on *some operation;*
► $\Delta \subseteq N\times N$ *is the set of arcs s.t.* $\Delta=\{(n_{op'_1},n_{op_1}),\cdots,(n_{op'_j},n_{op_j})\}$ *and for each* $i=1,\cdots,j$, *then* $(n_{op'_i},op_i,n_{op_i})\in\Delta$ *iff a* dependency exists *for* op_i, op_i depends on op'_i, $n_{op_i}=(op_i,N_{gen}(op_i))$, *and* $n_{op'_i}=(op'_i,N_{gen}(op'_i))$.

As already mentioned in Sect. 3, we need to "saturate" the automaton in order to complete it with respect to the possibility for the environment to directly provide input parameters. The result of this saturation step is called the *Saturated dependencies automaton*.

DEFINITION 3 (SATURATED DEPENDENCIES AUTOMATON).
Let $A_W=(N,\Delta)$ *be the* Dependencies automaton *of a WSDL interface* W, *the* Saturated dependencies automaton S_W *of* W *is the tuple* (N_{sat},Δ_{sat}) *where:*
► $N_{sat}=N\cup\{n_{Env}\}$ *s.t.* $n_{Env}=(Env,\mathrm{D})$ *is the environment node and* $\mathrm{D}=\{\bigstar\mapsto_t op.p \mid op\in \mathrm{Op_W}, p \text{ of type } t\}$;
► $\Delta_{sat}=\Delta\cup\Delta_{Env}$, $\Delta\cap\Delta_{Env}=\emptyset$, *s.t.* $\Delta_{Env}=\{(n_{Env},n_{op_1}),\cdots,(n_{Env},n_{op_j})\}$ *and for each* $i=1,\cdots,j$, *then* $(n_{Env},n_{op_i})\in\Delta_{Env}$ *iff a* dependency exists *for* op_i, *and* $n_{op_i}=(op_i,N_{gen}(op_i))$.

DEFINITION 4 (I/O DEPENDENCIES CHAIN).
Let $S_W=(N_{sat},\Delta_{sat})$ *be the saturated dependencies automaton of a WSDL interface* W, *a* I/O dependencies chain *of* S_W *is a* $c\in N^*_{sat}$ *defined in such a way that there exists* $m>0$, $n_0,\cdots,n_m\in N_{sat}$ *s.t.* $n_0=(Env,\mathrm{D})$, $c=\langle n_0 n_1\cdots n_m\rangle$, *and* $(n_0,n_1)\in\Delta_{sat},\cdots,(n_{m-1},n_m)\in\Delta_{sat}$.

Let $S_W=(N_{sat},\Delta_{sat})$ be a saturated dependencies automaton, given a node $n\in N_{sat}$, the set of I/O dependencies chains leading to n (and originating from the node of name Env) is denoted as $Ch(n)$. We denote the *normalization* of $Ch(n)$ with $\overline{Ch(n)}$ and it is defined as the set of traces of $Ch(n)$ without either loops (i.e., loop transitions) or cycles (i.e., cyclic paths).

Note that $\overline{Ch(n)}$ is a finite set, whereas $Ch(n)$ can be infinite.

Given $(op,D_{op})\in N_{sat}$ and $op\neq Env$, with $IP(op)$ we denote the set of instance pools for the operation of name *op*. That is $IP(op)=\{(p_1{:}v_1,\cdots,p_n{:}v_n)$ *s.t.* $p_1,\cdots,p_n$ *are input parameters of op and* $v_1,\cdots,v_n$ *are the values of* $p_1,\cdots,p_n$, *respectively*$\}$. With $SoapEnv$ we denote the set of all the SOAP messages that conform to the XML Schema of W. We denote the the oracle that we use for testing purposes as a function $\mathcal{O}racle : SoapEnv\longrightarrow\{regular,error\}$. In the following, we use a function $\mathcal{T}est_W : SoapEnv\longrightarrow SoapEnv$ that represents the execution of a test case (encoded as a SOAP message) on a WS implementing W. That is, it represents a WS operation invocation (i.e., the operation input message) retrieving another SOAP message as answer (i.e., the operation output message). We use also a function $\mathcal{R}esp2\mathcal{R}eqs : SoapEnv\times \mathrm{Op}_W\times \mathrm{Op}_W\longrightarrow SoapEnv^*$ that takes as input the response of the invocation of $op\in \mathrm{Op}_W$ and returns a tuple of requests for $op'\in \mathrm{Op}_W$ that depends on *op*. Thus, each of these requests is built by taking into account the dependencies stored in the node of *op*. Listing 2 is an operational description of the testing procedure that `StrawBerry` performs to produce the validated dependency automaton out of the saturated one. Note that this description is not the optimal algorithm with respect to computational load. However optimality is not the focus of this paper. This procedure exploits the $\mathcal{O}racle$, $\mathcal{T}est_W$, and $\mathcal{R}esp2\mathcal{R}eqs$ functions. The validated dependency automaton, as synthesized by our testing procedure, is defined by Def. 5. In Listing 2, given an operation $op\in \mathrm{Op_W}$, we denote the node of *op* in N_{sat} as $node(op)$. Furthermore, we denote a SOAP envelop message as either *soap* or $soap_i$ for some i.

Listing 2: `StrawBerry` testing procedure

```
Let S_W = (N,Δ) be the Saturated dependencies automaton
    of a WSDL interface W, perform the following steps:
Inizialization: mark every dependency in S_W as nonVisited;
  create an empty stack called Stack;
Step 1: while ∃ v=(op,D_op)∈N that stores a nonVisited
      uncertain dependency do
  while ∃ ch=⟨op_1···op_n op_{n+1}⟩∈Ch(op) (op=op_{n+1}) s.t.
       node(op_1)···node(op_{n-1}) store only certain
       dependencies and ∃ in node(op_n) a nonVisited
       dependency, op_{n+1} depends on, do
    if IP(op_1)≠∅ then produce soap_1 from IP(op_1);
      else produce soap_1 randomly for op_1;
    push Resp2Reqs(Test_W(soap_1),op_1,op_2) into Stack;
    set i to 2;
    while i<n+1 do
      foreach soap popped out from Stack do
        push Resp2Reqs(Test_W(soap),op_i,op_{i+1}) into Stack;
      set i to i+1;
  foreach soap popped out from Stack do
    if Oracle(Test_W(soap))=error then remove from
        node(op_n) all dependencies op_{n+1} depends on w.r.
        t. all the output parameters p of op_n that are
        involved in soap
      else mark as visited these dependencies;
  if node(op_n) stores no dependency op_{n+1} depends on then
      remove (op_n,op) from Δ
Step 2: if ∃ v=(op,D_op)∈N, op≠Env and v has no incoming
      arc then add (s_Env,v) to Δ, s_Env=(Env,D_Env), and
    add the corresponding certain dependencies to D_Env;
  foreach v=(op,D_op)∈N that stores an uncertain
      dependency do
  foreach op' that depends on op w.r.t. an uncertain
      dependency do
    produce soap randomly for op';
      if Oracle(Test_W(soap))=error then mark as certain
          all dependencies in node(op), op' depends on;
Step 3: foreach v=(op,D_op)∈N s.t. op ≠ Env do
  remove all the uncertain dependencies from D_op;
  if D_op=∅ and v has no incoming arc then remove v.
```

DEFINITION 5 (VALIDATED DEPENDENCY AUTOMATON).
The Validated dependency automaton V_W *of a WSDL interface* W *is the pair* (N,Δ) *that holds the following properties:*
• $\forall n\in N$: $\exists op\in \mathrm{Op_W}$: $n=(op,D_{op})$ *and either* D_{op} *contains only* certain *dependencies or it is empty*;
• $\forall n\in N$: $Ch(n)\neq\emptyset$;
• $\forall d\in\Delta$: $\exists op,op'\in \mathrm{Op_W}$: $d=((op,D_{op}),(op',D_{op'}))\wedge op.p\not\mapsto_t op'.p'\in D_{op}$.

From the *Validated dependency automaton*, the transformations specified in Def. 7 produce a *Behavior protocol automaton*.

Let $V_W=(N,\Delta)$ be the *Validated dependency automaton* of a WSDL interface W, with $\mathrm{ioDS}(V_W)$ we denote the I/O dependency set of V_W and with $\mathrm{Op}(V_W)$ the set of operation names for V_W. $\mathrm{ioDS}(V_W)$ corresponds to the set of I/O dependencies stored in the nodes of V_W. Note that they are all certain dependencies. $\mathrm{Op}(V_W)$ corresponds to the set of operation labels stored in the nodes of V_W, including `Env`. Starting from V_W `StrawBerry`

produces a table T_{op} for each $op \in \mathrm{Op}(V_W)$ different from Env. T_{op}=$\{(o_1,\cdots,o_n)\in \mathrm{Op}(V_W)^n$ *s.t. n is the number of input parameters of op and for each parameter p of op, an operation* o_i *exists s.t.* $o_i.p' \nmapsto_t op.p \in ioDS(V_W)\}$.

By taking into account each T_{op}, `StrawBerry` produces a set Υ of sets of operations for which no mutual dependency exists. Each set of operations in Υ corresponds to a connected component in the behavior protocol automaton to be synthesized.

For example, $\{op,op'\}\in\Upsilon$ means that neither $op.p \nmapsto_t^* op'.p'$ nor $op'.p' \nmapsto_t^* op.p$ hold for any p, p', and we say that op and op' are independent. Thus four states, s_1,s_2,s_3, and s_4, and four transitions, (s_1,op,s_2), (s_2,op',s_4), (s_1,op',s_3), and (s_3,op,s_4), are produced in the behavior protocol automaton. In general, if $op_1,\cdots,op_n$ ($n>1$) are independent, then $n+(\sum_{i=1}^{n-1}\frac{n!}{i!})+2$ states are generated and sequences of transitions labeled with $op_1,\cdots,op_n$ are produced among these states in order to build all the linearizations modeling the interleaving of $op_1,\cdots,op_n$. If $\{op\}\in\Upsilon$, the produced connected component is represented by the transition $(s1, op, s2)$ and the states $s1$ and $s2$. Note that each of these connected components has a source state and a sink state. For the sake of presentation, in Def. 7 we use a function, CCB (Connected Component Builder), that takes as input V_W and produces the set $\{k_1,\cdots,k_h\}$ of above discussed connected components. We denote with k_i^{source} and with k_i^{sink} the source state and the sink state of k_i, respectively. Def. 7 uses the definition of trace for a behavior protocol automaton (see Def. 6).

DEFINITION 6 (TRACE).
Let I_W=(S,F,s_0,A,Δ) *be a behavior protocol automaton, a* trace *of* I_W *is a* $t\in A^*$ *defined in such a way that there exist* $n>0$, $s_0,\cdots,s_n\in S$ *such that* $t=\langle o_1 o_2\cdots o_n\rangle$ *and* $(s_0,o_1,s_1)\in\Delta,\cdots,(s_{n-1},o_n,s_n)\in\Delta$.

Let I_W=(S,F,s_0,A,Δ) be a behavior protocol automaton, given a state $s\in S$, the set of traces leading to s (and originating from s_0) is denoted as $Tr(s)$.

DEFINITION 7 (BEHAVIOR PROTOCOL AUTOMATON). *Let* $V_W = (N,\Delta)$ *be the* Validated dependencies automaton *of a WSDL interface* W, *the* Behavior protocol automaton *of* W *is the tuple* (S,F,s_0,A,Δ') *where:*
▶ S= $\{s|s$ *is a state of a connected component* $k\in CCB(V_W)\}$.
▶ F= $\{s_F^1,\cdots,s_F^m\}$ *where each* s_F^i *is the sink state of a connected component built from operations that are not source operations of any dependency.*
▶ s_0= k^{source} *where* $k\in CCB(V_W)$ *is built from only* Env.
▶ A=$\{op(p_1,\cdots,p_m)$:$o_1,\cdots,o_k$ *s.t.* $op\in \mathrm{Op_W}$, $o_1,\cdots,o_k$ *are output parameters of op and* $p_1,\cdots,p_m$ *are input parameters of op*$\}$.
▶ $\Delta'\subseteq S\times A\times S$, $\Delta'=\Delta_{cc}\cup\Delta''\cup\Delta_{loop}$, *and* $\Delta_{cc}\cap\Delta''\cap\Delta_{loop}=\emptyset$, *where:*
▼ Δ_{cc} *is the union set of the sets of transitions of each connected component* $k\in CCB(V_W)$.
▼ Δ''=$\{(s,l,s')$ | *there exist* $op\in\mathrm{Op}(V_W)$ *and* $k\in CCB(V_W)$ *such that:* $s'=k^{sink}$ *and* k *contains a transition labeled with op; and for each* $tr=\langle op_1\cdots op_n\rangle\in Tr(s)$ *and each* $(o_1,\cdots,o_m)\in T_{op}$ *s.t.* o_i *contained in* tr, *then* $l=op(p_1,\cdots,p_m)$:$out_1,\cdots,out_k$ *where* $out_1,\cdots,out_k$ *are output parameters of op, for each* $i=1,\cdots,m$ *then* p_i *is an output parameter of* o_i, *and* $o_i.p_i \nmapsto_t op.p \in \mathrm{ioDS}(V_W)$ *for some p that is input parameter of op*$\}$.
▼ Δ_{loop}=$\{(s,l,s)$ | *for all* $s\in S$, $op\in\mathrm{Op_W}\setminus\mathrm{Op}(V_W)$, $l=op(p_1,\cdots,p_m)$:$o_1,\cdots,o_k$ *where* $o_1,\cdots,o_k$ *are output parameters of op and* $p_1,\cdots,p_m$ *are input parameters of op*$\}$.

5. THE AMAZON E-COMMERCE SERVICE CASE STUDY

In this section, we show the results of the application of `StrawBerry` to an existing WS, that is the *Amazon E-Commerce Service* (AECS). AECS is part of the Amazon Associates WS suite[5]. The aim of this section is to show the applicability of `StrawBerry` to a complex WS that is well-known and widely used by practitioners. Moreover, AECS is well-documented. Thus it allowed us to validate that the concepts underlying `StrawBerry` and the choices we made to realize them are reasonable in practice.

Operation	***certain*** **after 1.2**	***uncertain*** **after 1.2**	***uncertain*** **after 4.1**	***uncertain*** **after 4.2**
Help	2	2358	0	0
ItemSearch	44	2838	18	0
ItemLookup	44	2838	14	0
BrowseNodeLookup	2	1572	0	0
ListSearch	46	11222	8	0
ListLookup	46	11222	4	0
CustomerContentSearch	46	15676	4	0
CustomerContentLookup	46	15676	8	0
SimilarityLookup	44	2838	12	0
SellerLookup	2	5502	10	0
CartGet	2	4978	32	0
CartAdd	2	4978	24	0
CartCreate	2	4978	16	0
CartModify	2	4978	32	0
CartClear	2	4978	16	0
TransactionLookup	2	5502	6	0
SellerListingSearch	2	8908	10	0
SellerListingLookup	2	8908	12	0
TagLookup	46	11222	38	0
VehicleSearch	2	1310	20	0
VehiclePartSearch	46	3886	4	0
VehiclePartLookup	46	3886	4	0
MultiOperation	90	35020	75	0
Total:	568	175274	367	0

Table 2: Summary of the AECS case study results

We focus on discussing the collected results rather than on detailing the execution steps of `StrawBerry` to AECS. This would be impractical due to the size of AECS in terms of I/O data dependencies. Starting from the AECS WSDL[6], `StrawBerry` performs the steps described in Sect. 3.3 producing the results discussed below.

Step 1.1: `StrawBerry` discovers that AECS exports 23 operations and elicits 187894 dependencies. Table 2 summarizes the discovered data; the operations are listed as they appear in the WSDL of AECS.

Step 1.2: heuristics 1 and 2 allow `StrawBerry` to promote 568 dependencies as *certain* (40 and 528, respectively). The AECS reference guide allows us to enable heuristic 3 since it reports that all the operation parameters of type `Errors` are used to encode error answers. Thus, Heuristic 3 removes 12052 dependencies. In Table 2 we show how the 568 discovered *certain* dependencies are distributed among the AECS operations. At the end of this step, there remain 175274 *uncertain* dependencies.

Step 2.1: `StrawBerry` generates 23 nodes, one for each operation, and each of them stores a number of *certain* and *uncertain* dependencies as reported by the first three columns of Tab. 2.

Step 2.2: `StrawBerry` generates a dependency automaton that has 23 nodes and 529 arcs. The size of this automaton prevents us to graphically show it. However, in Fig. 7, we show an excerpt from

[5]Amazon Associates: http://aws.amazon.com/associates/
[6]AECS WSDL: http://webservices.amazon.com/AWSECommerceService/AWSECommerceService.wsdl

the behavior protocol automaton constructed by StrawBerry at the end of the process.

Step 2.3: the previous automaton is saturated by adding the Env node and 23 arcs, each of them from Env to another node. Env stores 350 *uncertain* dependencies.

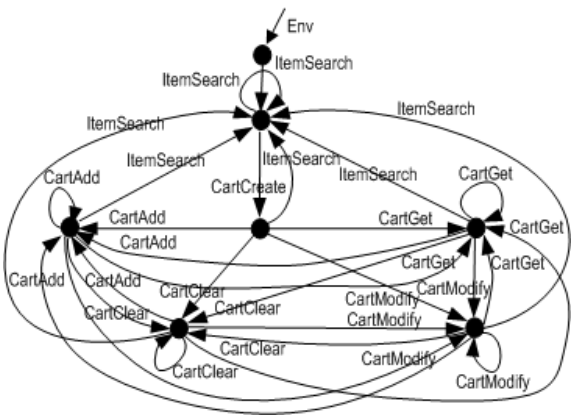

Figure 7: An excerpt from the behavior protocol of AECS

Activity 3: we build an instance pool in order to generate, by means of StrawBerry, the SOAP envelop messages for testing AECS. We need to provide SOAP test messages with a pair of unique identifiers that are required, for security purposes, by each operation. These identifiers are provided by Amazon after the *Amazon Associate* registration process. Furthermore, we can add instance pools related to meaningful Amazon items, e.g., some author names as shown in Tab. 1.

Step 4.1: the previously generated test cases are used by StrawBerry to prune the set of 175274 *uncertain* dependencies obtained after step 1.2. After step 4.1, in which we test each of them, only 367 *uncertain* dependencies survive, distributed among the operations as shown in the fourth column of Tab. 2. Note that, already after step 4.1, we can have some operations (Help and BrowseNodeLookup) for which no *uncertain* dependency survives.

Step 4.2: the survived 367 dependencies are confirmed as *certain* by the tests in step 4.2 and hence there is no *uncertain* dependency to be solved in step 4.3. Thus, StrawBerry directly performs Activity 5.

Activity 5: starting from the validated dependency automaton, as obtained after the execution of step 4.2, StrawBerry synthesizes the behavior protocol automaton of AECS. The validated dependency automaton has 24 nodes and 288 arcs. In Fig. 7, we show an excerpt concerning all the "item search" and "cart management" operations of AECS. For the sake of readability we omit the data parameters in the operation labels. Furthermore, for each state there are other incoming and outgoing transitions from and to states that do not appear in the figure. By looking at Fig. 7 one could think that one specific state is corresponding to one specific operation; however as already seen in Fig. 6, this does not hold in general.

We performed some ad hoc validation of the synthesized automaton. We checked that all the results described above and the synthesized protocol match with what is described in the AECS API reference[7]. Moreover we further validated the synthesized behavior protocol of AECS through a web client provided by Amazon[8]. Thus, we empirically verified that StrawBerry produces a realistic model for AECS.

[7]AECS API reference: http://awsdocs.s3.amazonaws.com/ECS/latest/aaws-dg.pdf

[8]http://www.awszone.com/scratchpads/index.aws

Besides showing the effectiveness of StrawBerry, this case study highlights that, even when the necessary information is available, the hand-made provisioning of the behavior protocol is a difficult and error prone task.

6. RELATED WORK

Several authors have recently addressed the problem of deriving a behavioral model from an implemented system. We discuss here some of these works with respect to the StrawBerry method.

In [9], the authors describe a technique, called GK-Tail, to automatically generate behavioral models from (object-oriented) system execution traces. GK-Tail assumes that execution traces are obtained by monitoring the system through message logging frameworks. For each system method, an Extended Finite State Machine (EFSM) is generated. It models the interaction between the components forming the system in terms of sequences of method invocations and data constraints on these invocations. The correctness of these data constraints depends on the completeness of the set of monitored traces with respect to all the possible system executions that might be infinite. Furthermore, since the set of monitored traces represents only positive samples of the system execution, their approach cannot guarantee the complete correctness of the inferred data constraints. 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. Finally note that StrawBerry is an extra-procedural method, whereas GK-Tail is intra-procedural. In fact we synthesize a model of the possible interactions between the WS and its environment, whereas they synthesize an intra-system interaction model.

The work described in [8] (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 proceeds in two main stages: first, SPY infers a partial model of the considered Java class; second, this partial model is generalized to deal with data values beyond the ones specified by the given instance pools. The model generalization is based on two assumptions: (i) the value of method parameters does not impact the implementation logic of the methods of a class; (ii) the behavior observed during the partial model inference process enjoys the so called "continuity property" (i.e., a class instance has a kind of "uniform" behavior). In our context, we cannot rely on the previously mentioned assumptions.

The approach described in [16], and implemented by Jadet, analyzes 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, Jadet is a white-box method. Furthermore, as it is for the work described in [9], Jadet focuses on modeling objects from the point of view of single methods that is a goal different from ours. The work described in [15] (i.e., OP-Miner) is very similar to Jadet. Differently from our work, it is a white-box approach. Java code is analyzed to infer the sequence of operations an object variable goes through before being used as a parameter. In general, this is slightly similar to what StrawBerry synthesizes but, in practice, analogously to what Jadet does, this is done by looking at each single method. In this sense the analysis performed by OP-Miner (and Jadet) is intra-procedural, whereas our approach is extra-procedural.

The authors of [5] present an approach for inferring state ma-

chines 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 [11] describe a learning-based black-box testing approach in which the problem of testing functional correctness is reduced to a constraint solving problem. The testing phase of our approach shares some ideas with the approach described in [11]. That is, through black-box testing, we refine an approximated data-flow model in order to prune fake I/O dependencies. However, we do not use *function approximation theory* and our coverage criterion is established by looking at the inferred I/O dependencies.

7. CONCLUSIONS AND FUTURE WORK

In this paper we have presented the `StrawBerry` method. It takes as input a WSDL description, matches by type the input and output parameters of its operations, applies some graph synthesis and heuristics, and going through a testing phase, eventually synthesizes what we have called the *Behavior Protocol* automaton.

In our view, `StrawBerry` fulfills a widespread exigency, that is to get more semantic information for a WS, whereby the current practice is to publish only its signature. A behavioral model is highly desirable both for a client that needs to understand how a WS should be used, and for a WS integrator to properly architect WSs composition.

The method that we propose is practical and realistic in that it only assumes: (i) the availability of the WSDL; (ii) the possibility to derive a partial oracle that can distinguish between *regular* and *error* answers. This oracle is needed in the testing stage to confirm or reject uncertain dependencies. In this paper we have taken assumption (ii) in strict sense, in that we have assumed the existence of this oracle, i.e., that the available WS information allows a tester to recognize when a test outcome is an error message. This was indeed true for the AECS case study. In future work, we intend to investigate if and how assumption (ii) could be relaxed, and, where such a partial oracle does not exist, the deterministic testing steps could be replaced by a statistical testing session.

We have started to show, through its application to a real world case study, that the method is viable, and that it nicely converges to a realistic automaton. We obviously need to carry out more empirical investigation to convey such preliminary evidences into a real quantitative assessment of the method. However, the case study convinced us that the combination of heuristics and basic testing can work quite effectively. In particular, the introduction of heuristics for optimization seems interesting and we believe that it is the first direction to push further to reduce the testing effort in the subsequent steps. For AECS the execution of each test case took a time in the order of 10^{-2} sec., but the total number of test cases was in the order of 10^5, summing up to few hours of testing. This is perfectly acceptable for off-line analysis, but is certainly not usable for dynamic analysis.

Indeed, the long term goal of this research is to support dynamic WS composition. Here we have developed a first step of deriving a behavioral model from the WSDL of a WS. Our future work will investigate how to compose and validate several derived behavior protocol automata to achieve a desired composite system.

Acknowledgments

This work is partly supported by the EU IST CONNECT (http://connect-forever.eu/) No 231167 of the FET - FP7 program and the Italian PRIN d-ASAP projects. We thank Amleto Di Salle for the useful discussions on WS development technologies, Cesare Bartolini and Francesca Lonetti for their support concerning the use of WS-TAXI.

8. REFERENCES

[1] BPEL4WS: Business Process Execution Languages for Web services v1.1 specification. http://download.boulder.ibm.com /ibmdl/pub/software/dw/specs/ws-bpel/ws-bpel.pdf.

[2] CORDIS, ICT, Programme: Service and Software Architectures, Infrastractures and Engineering (SSAI). http://cordis.europa.eu/fp7/ict/ssai/home_en.html.

[3] WSDL: Web Services Description Languages v1.1 spec. http://www.w3.org/tr/2001/note-wsdl-20010315.

[4] 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.

[5] 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.

[6] A. Brogi and R. Popescu. Automated generation of BPEL adapters. In *ICSOC 2006, Chicago, USA*, 2006.

[7] D. Calvanese, G. D. Giacomo, M. Lenzerini, M. Mecella, and F. Patrizi. Automatic Service Composition and Synthesis: the Roman Model. *IEEE Data Eng. Bull.*, 31(3):18–22, 2008.

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

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

[10] A. Marconi, M. Pistore, and P. Traverso. Automated Composition of Web Services: the ASTRO Approach. *IEEE Data Eng. Bull.*, 31(3):23–26, 2008.

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

[12] T. Melliti, P. Poizat, and S. B. Mokhtar. Distributed Behavioural Adaptation for the Automatic Composition of Semantic Services. In *FASE 2008, LNCS 4961, Springer*.

[13] J. Pathak, S. Basu, R. R. Lutz, and V. Honavar. MOSCOE: an Approach for Composing Web Services through Iterative Reformulation of Functional Specifications. *Int. Journal on Artificial Intelligence Tools*, 17(1):109–138, 2008.

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

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

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