Automated Inference of Models for Black Box Systems Based on Interface Descriptions*

Maik Merten[1], Falk Howar[1], Bernhard Steffen[1],
Patrizio Pellicione[2], and Massimo Tivoli[2]

[1] Technical University Dortmund, Chair for Programming Systems,
Dortmund, D-44227, Germany
{maik.merten,falk.howar,steffen}@cs.tu-dortmund.de
[2] Università dell'Aquila, Dipartimento di Informatica, Via Vetoio, L'Aquila, Italy
{patrizio.pelliccione,massimo.tivoli}@univaq.it

Abstract. In this paper we present a method and tool to *fully automatically* infer data-sensitive behavioral models of black-box systems in two coordinated steps: (1) syntactical analysis of the interface descriptions, here given in terms of WSDL (Web Services Description Language), for instantiating test harnesses with adequate mappers, i.e., means to bridge between the model level and the concrete execution level, and (2) test-based exploration of the target system by means of active automata learning. The first step is realized by means of the syntactic analysis of `StrawBerry`, a tool designed for syntactically analyzing WSDL descriptions, and the second step by the `LearnLib`, a flexible active automata learning framework. The new method presented in this paper (1) overcomes the manual construction of the mapper required for the learning tool, a major practical bottleneck in practice, and (2) provides global behavioral models that comprise the data-flow of the analyzed systems. The method is illustrated in detail along a concrete shop application.

1 Introduction

Documentation of IT-systems is, in well-known practice, usually found to be incomplete and inaccurate or otherwise lacking. This can be a major obstacle for continued development of affected systems, where, e.g., extensions to the systems should not lead to regressions: without an informative specification of the expected behavior it is difficult to ensure that all relevant regressions have been discovered during testing and remedied before product deployment.

Inaccurate specifications also create major challenges when trying to connect remote Networked Systems (NSs). Thus making such specifications precise is one of the major challenges of the CONNECT project [4], which, even more ambitiously, aims at creating an infrastructure where networked connectors can be synthesized fully automatically.

In this paper we present a method and tool to *fully automatically* infer dataflow-sensitive behavioral models of black-box systems based on interface descriptions in WSDL, the Web Services Description Language. This solves the problem of deriving

* This work is partially supported by the European FP7 project CONNECT (IST 231167).

T. Margaria and B. Steffen (Eds.): ISoLA 2012, Part I, LNCS 7609, pp. 79–96, 2012.

system specifications of black box systems adequate to, e.g., serve as a basis for the connector synthesis of the CONNECT platform. This is done in two coordinated steps: (1) syntactical analysis of the interface descriptions, here given in terms of WSDL, for instantiating test harnesses with adequate mappers, i.e., means to bridge between the model level an the concrete execution level, and (2) test-based exploration of the target system by means of active automata learning.

The first step is realized by means of `StrawBerry`, a tool designed for syntactically analyzing WSDL descriptions, and the second step by the `LearnLib`, a flexible active automata learning framework. The combination of the two tools (1) overcomes the manual construction of the mapper required for the learning tool, a major practical bottleneck in practice, and a show stopper for automated model generation, and (2) provides global behavioral models that comprise the data-flow of the analyzed systems. Thus it is unique in combining the general applicability of `StrawBerry`, which simply requires WSDL interfaces, with the ability of active automata learning to infer data-sensitive behavioral models.

The presentation of our method is accompanied by a discussion along a concrete shop application, which illustrates the main features and highlights the essence of dataflow sensitive modeling.

The paper is structured as follows. Sect. 2 presents a motivating and running example. Sect. 3 provides background information on `StrawBerry` and Sect. 4 provides information on `LearnLib`. The integration of syntactic interface analysis and automata learning is discussed in Sect. 5, for which results are provided and discussed in Sect. 6. Related work is discussed in Sect. 7, before Sect. 8 closes with our conclusions and directions to future work.

2 Motivating Example

The explanatory example that we use in this paper is a web service (WS) called `EcommerceImplService`. This service simulates a small e-commerce service, where clients can open a session, retrieve a list of products, add products to a shopping cart and finally conclude buying the items previously added to the cart. The following operations are defined in the WSDL interface description:

• `openSession`: this operation is used by registered users to login into the WS. The operation gets the `username` and `password` as input and returns a `session`. `session` is a complex type composed of a session `id` and `creationTime`.

Input data	Output data
`user: string;`	`return: session;`
`password: string;`	

• `destroySession`: this operation gets as input a `session`, destroys this session, and returns a string denoting success.

Input data	Output data
`session: session;`	`return: string;`

• `getAvailableProducts`: this operation gets no inputs and returns `productArray`, i.e., a list of products, where a `product` is a complex type composed of the product `id`, its `description`, and its `price`.

Input data	Output data
	`return: productArray;`

• `emptyShoppingCart`: this operation gets as input a `session`, empties the shopping cart, and returns the current `session`.

Input data	Output data
`session: session;`	`return: session;`

• `getShoppingCart`: this operation gets as input a `session` and returns the current `shoppingCart`. `shoppingCart` is a complex type composed of a cart `id`, a list of `products`, and the `price`.

Input data	Output data
`session: session;`	`return: shoppingCart;`

• `addProductToShoppingCart`: this operation gets as input a `session` and a `product`, adds the product to the shopping cart, and returns the current `session`.

Input data	Output data
`session: session;`	`return: session;`
`product: product;`	

• `buyProductsInShoppingCart`: this operation gets as input a `session`, buys the array of products contained into the shopping cart and returns this array.

Input data	Output data
`session: session;`	`return: productArray;`

The particular implementation of this service has the following three semantic properties, which we will use for the illustration of our method. We will see that `StrawBerry` fails to detect all of them, but that the integrated approach detects them all:

- The operation `buyProductsInShoppingCart` will only successfully conclude if the shopping cart connected to the current session is not empty. Otherwise an error will be raised.
- In contrast, the operation `emptyShoppingCart` will return successfully even if the shopping cart was empty already, as long as a valid session is provided.
- The shopping cart is emptied on successful invocations of `buyProductsInShoppingCart`.

This behavior was modeled to reflect actual web shops. That is, web shops usually do not allow for empty orders, as sending, e.g., empty packages to customers will nonetheless inflict costs. Performing a clearing operation on an empty shopping cart, however, is not hurtful. Upon concluding a purchase, customers will expect a "fresh" shopping cart, so they can resume shopping without having to worry about potentially shopping items twice.

There are several reasons why we chose to use a simulated e-commerce service over, e.g., an actual e-commerce service available on the Internet. First, public e-commerce services usually do not offer an experimental mode where orders will not actually result in costly deliveries and extensive test runs during the extrapolation of the service will not be interpreted as, e.g., a denial of service attack. Second, the simulated e-commerce service is comparatively small, which allows for easy comparison of the extrapolated models with the actual implementation.

3 StrawBerry

By taking as input a WSDL of a WS (Web Service), `StrawBerry` derives in an automated way a partial ordering relation among the invocations of the different WSDL operations. This partial ordering relation is represented as an automaton that we call *Behavior Protocol automaton*. It models the interaction protocol that a client has to follow in order to correctly interact with the WS. This automaton also explicitly models the information that has to be passed to the WS operations. The behavior protocol is obtained through synthesis and testing stages. The synthesis stage is driven by syntactic interface analysis (aka 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.

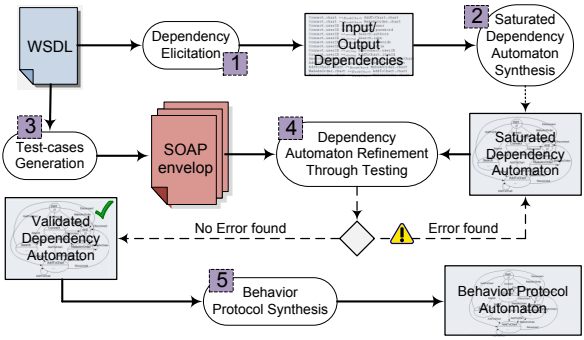

Fig. 1. Overview of the `StrawBerry` method

`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 [15,24,25]. 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 [6].

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

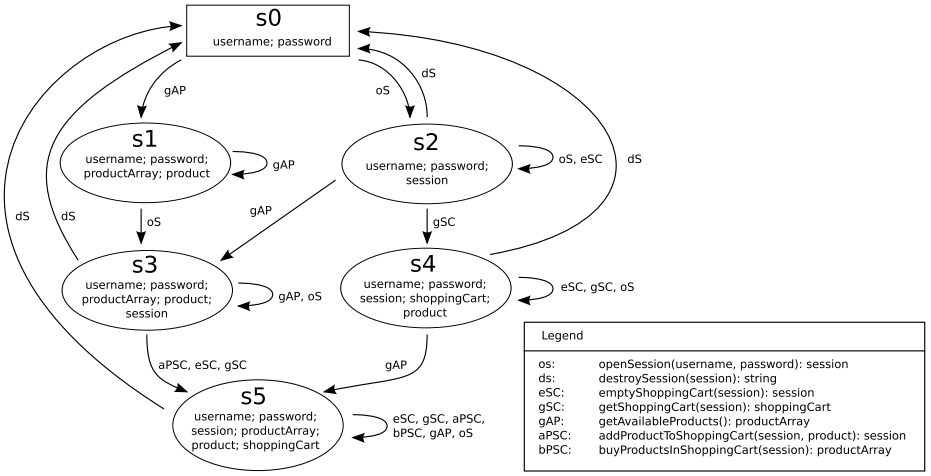

Fig. 2. Model created by `StrawBerry`. The edge labels are abbreviated for improved readability.

operation. This model is completed by applying a *saturation rule*. This rule adds new dependencies that model the possibility for a 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 the operations to be tested, according to the WSDL of the WS. In `StrawBerry` tests are generated from the WSDL and aim at validating whether the synthesized automaton is a correct abstraction of the service 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, these false dependencies are deleted from the automaton.

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.

The primary result of `StrawBerry` used in the subsequent learning phase is the set of validated "chains" of data dependencies.

`StrawBerry` at Work: referring to the example described in Section 2, Figure 4 shows states of the dependencies automaton produced by `StrawBerry`. Each state

contains dependencies that each operation has with other operations. Dependencies marked with ✓ represent dependencies that are validated by testing activities. Figure 2 shows the obtained behavioral automaton. 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* [11], is available. Nodes of the behavioral automaton contain the matured "knowledge", i.e., the data that are provided with the instance pool or that are obtained as result of previously invoked operations. The `S0` state contains only information that comes from the instance pool, i.e., `username` and `password`. In `S0` only `openSession` and `getAvailableProducts` can be invoked. Once invoked the `openSession` operation, the service reaches the state `S2` in which `session` is available, since it is returned by the `openSession` operation. Similarly, by executing `getAvailableProducts` the service reaches the state `S1` in which both `productArray` and `product` are available since `productArray` is the return value of `getAvailableProducts` and `product` is nested into the complex type `productArray`.

Let us now focus on the state `S5`; in this state each operation can be invoked. Indeed this automaton does not represent an accurate model for `EcommerceImplService`. In particular the semantic properties introduced above are not revealed. For instance, `buyProductsInShoppingCart` might fail when the shopping cart is empty. In other words, there exist a sequence of operations that might lead to `S5` with an empty cart. The lack of behavioral information in the produced model can be attributed to the fact that web service interfaces are not concerned with describing behavioral aspects and thus provide incomplete information to any analysis approach merely focusing on interfaces. As discussed in the following sections, the approach that we present in this paper overcomes this limitation.

4 LearnLib and Active Automata Learning

`LearnLib` is a framework for automata learning and experimentation. Active automata learning tries to automatically construct a finite automaton that matches the behavior of a given target automaton on the basis of active interrogation of target systems and observation of the produced behavior.

Active automata learning originally has been conceived for language acceptors in the form of deterministic finite automata (DFAs) (cf. Angluin's L^* algorithm [3]). It is possible, however, to apply automata learning to create models of reactive systems instead. A more suited formalism for this application are Mealy machines:

Definition 1. *A Mealy machine is defined as a tuple* $\langle Q, q_0, \Sigma, \Omega, \delta, \lambda \rangle$ *where*

- Q *is a finite nonempty set of* states *(be* $n = |Q|$ *the size of the Mealy machine),*
- $q_0 \in Q$ *is the* initial state*,*
- Σ *is a finite* input alphabet*,*
- Ω *is a finite* output alphabet*,*
- $\delta : Q \times \Sigma \to Q$ *is the* transition function*, and*
- $\lambda : Q \times \Sigma \to \Omega$ *is the* output function.

Intuitively, a Mealy machine evolves through states $q \in Q$, and whenever one applies an input symbol (or action) $a \in \Sigma$, the machine moves to a new state according to $\delta(q, a)$ and produces an output according to $\lambda(q, a)$.

In the context of reactive systems, the input alphabet contains actions which can be executed on the target system, while the output alphabet is determined by the output the system produces in response to the executed input actions.

Mealy machines are deterministic and thus are not a fitting modeling approach for, e.g., systems with truly erratic behavior, such as slot machines. However, many (if not most) systems serving a specific purpose are deterministic in nature, i.e., provided with a fixed set of inputs applied to a preset internal state, these systems will always produce the same output. Spurious errors (e.g., due to errors in communication) can be detected and eventually corrected by means of repeated experimentation.

When employed to create models in the form of Mealy machines, active automata learning employs two distinct types of queries to gather information on the System Under Learning (SUL):

- Membership Queries (MQs) retrieve behavioral information of the target system. Consisting of traces of system stimuli (each query $mq \in \Sigma^*$), MQs actively trigger behavioral outputs which are collected and analyzed by the learning algorithm. MQs are used to construct a hypothesis, which is subject of a verification by a second class of queries, the equivalence queries.
- Equivalence Queries (EQs) are used to determine if the learned hypothesis is a faithful representation of the target system. If the equivalence oracle handling the EQ finds diverging behavior between the learned hypothesis and the target system a counterexample $ex \in \Sigma^*$ will be produced, which is used to refine the hypothesis after restarting the learning process.

With those two query types, learning algorithms, such as $L^*_{i/o}$ [17], create minimal automata models, i.e., the learned result never contains more states than the minimized representation of the target system, and also guarantee termination with an accurate learned model.

It is worth noting that while MQs are relatively straightforward to employ on actual systems by execution of test runs, EQs pose a more challenging problem: while systems will readily produce output in response to input as normal mode of operation, they usually will neither confirm nor disprove a learned hypothesis in a direct manner. This is easy to see, as systems usually do not possess a formal model of their inner-workings fit for comparison. Thus, in practice, EQs can only be approximated, e.g., by executing additional test runs by means of MQs. Employing approximated EQs does impact the statement on correctness presented above: while the learned model will still be minimal, its accurateness is no longer guaranteed. In certain applications, however, it is possible to construct perfect EQs by employing MQs, e.g., if an upper bound on system size in terms of states is known. For the experiments presented in this paper, however, a simple approximation was used that generates random test runs.

`LearnLib` contains several learning algorithms fit for learning reactive systems, including EQ approximations, embedded in a flexible framework.

In practice, to learn concrete reactive systems, a *test-driver* has to translate the generated queries composed of abstract and parameterized symbols into concrete system interaction and conduct the actual invocations. In turn, the produced (concrete) system output has to be gathered and translated into abstract output symbols. Figure 3 shows the essential components of such a test-driver, embedded into a learning setup.

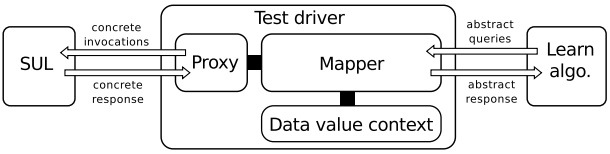

Fig. 3. Schematic view of a test driver for learning a reactive system

- A *mapper* is responsible for translating abstract queries generated by the learning algorithm into concrete queries comprised of actions that can be executed on the SUL. For parameterized actions, fitting valuations have to be inserted. Mappers are discussed, e.g., in [14].
- To fill in values for parameterized actions, a *data value context* maintains a set of value instances, that can be stored, retrieved and updated by the mapper.
- The *proxy* maintains a connection to the SUL and interacts with the SUL on behalf of the test-driver, using the concretized parameterized actions created by the mapper. Invocation results are gathered and returned to the mapper, which creates fitting abstract output symbols. For remote services which deliver an interface description in a standardized format (for instance, WSDL), such proxies can often be generated using specialized tools.

`LearnLib` at Work: `LearnLib` employs active automata learning algorithms that belong to the family of L^*-like algorithms. Models are constructed by gathering observations triggered by active infusion of test queries. This approach works without having any knowledge on the syntactic structure of the system's interface. In fact, queries are assembled from a provided alphabet without any notion of syntactic correctness, although having such a notion can speed up the learning process by filtering not well-formed queries (e.g., in the test driver) and only executing syntactically correct on the target system. Even the alphabet from which queries are constructed may be comprised of arbitrary bit-strings, which, of course, do not bode well regarding the chances of creating an insightful model.

To be able to learn models for systems on a sophistication level of the discussed example system, it is necessary to handle data dependencies of the actions to be invoked. This means that the abstract alphabet symbols in fact are parameterized, with fitting valuations being inserted at runtime and returned data values being retained as needed. This is done in the test-driver by the mapper component with data values being organized in a data value context, as discussed in Sect. 4.

In current practice, both the learning alphabet and the according mapper are constructed manually. This can be a time-consuming task, with, for example, more than a

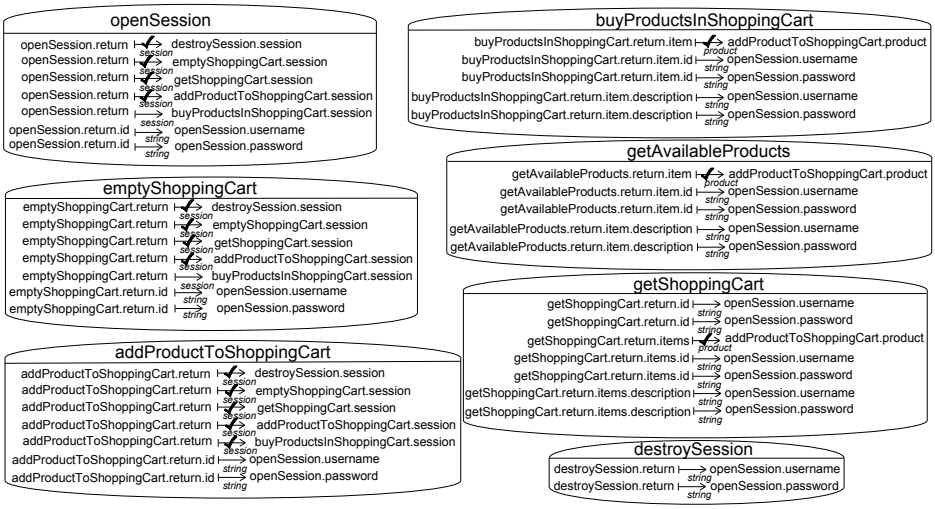

Fig. 4. States of the dependencies automaton produced by `StrawBerry`

quarter of the total effort being attributed to theses tasks in [23]. This manual approach of creating automata learning setups induces clear limitations on where automata learning can be employed. For example, this is unsustainable in scenarios where behavioral models are to be learned automatically for a wide range of systems, which is a requirement, e.g., for automated connector synthesis.

A central bottleneck of current practice is that the test-driver components such as the mapper must be constructed manually for any specific SUL. This is overcome by our approach which uses a generic mapper that is automatically instantiated with information derived from the syntactic interface analysis performed by `StrawBerry`.

5 The Integrated Approach

The integrated approach that is proposed in this paper solves limitations of both `StrawBerry` and `LearnLib`. Conceptually, the new solution integrates learning techniques with syntactic analysis that helps identifying potential dependencies between input and output parameters of different service operations. The integrated approach is an automata learning method, which is automated by a tool, that is realistic since it requires as input only a WSDL interface. As far as we know this is the only method with such minimal input assumption. It is worthwhile to note that, although `StrawBerry` shares the same minimal input assumption, it does not perform automata learning. In fact, it performs a totally different approach (based on data analysis and testing) that is less accurate than automata learning. Accuracy is a key aspect related to the behavioral model inference problem.

As typical usage scenario of the integrated approach let us imagine that a user needs to understand the behavior automaton of an existing black-box WS, such as the Amazon E-Commerce Service (AECS) as shown in [7]. The overall information that the user has

to provide are: (i) the URL of the service to be learned; (ii) predetermined data values for an instance pool; (iii) alphabet symbols which refer to parameterized actions on the target system; and (iv) parameters and return variables for each alphabet symbol. Even though in this paper we consider a mock-up service built in house with the aim of carrying out meaningful validation, this usage scenario points it out that our approach is realistic in the sense that it can be applied to third-parties black-box services. As it is usual in the current practice of web-services development, service providers give access to a testing version of the same service that allows developers to extensively test web-services while avoiding negative side effects on the availability of production services. For instance, this is the case for the Amazon case study described in [7] and for other well-known third-parties services, such as PayPal.

The integrated approach enhances `LearnLib` with syntactic analysis that extracts from running services a WSDL enriched with explicit I/O data dependencies. In `LearnLib`, and in general in active learning, this information is assumed to be provided by users and to be part of the learning setup. However, producing this information would be complex and for sure tedious.

Glue connectors have been realized to enable `LearnLib` to take as input the dependency analysis results produced by syntactic analysis. More precisely, glue connectors have been realized to take as input the enriched WSDL and to allow for the automatic construction of a mapper required for the learning tool to bridge between the model level (abstract alphabet symbols) and the concrete execution level (concrete actions outfitted with live data values and return values of invocations).

The syntactic analysis of the integrated approach, which is needed to allow the construction of an alphabet and a mapper accounting for data flow concerns, is inherited by `StrawBerry`. This part of `StrawBerry`, i.e. activities 1 and 2 referring to Figure 1, produces an automaton that is handed over to `LearnLib` in form of an artifact called *setup specification* (an overview is given in Figure 5).

We recall that Figure 4 shows the states of the saturated dependencies automaton produced by `StrawBerry`'s syntactic analysis and syntactic dependencies that each operation has with the other operations. This information is used in the integrated ap-

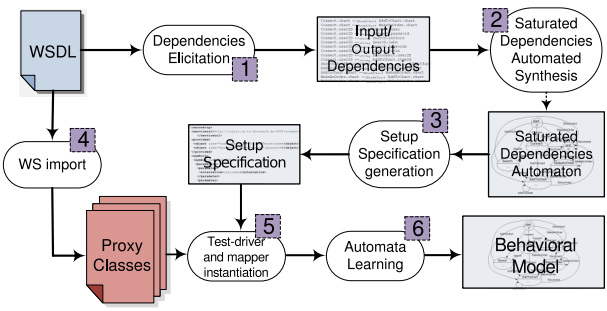

Fig. 5. Integration of `StrawBerry`' syntactic analysis (steps 1-2) and `LearnLib` (steps 4, 5, and 6). Step 3 is a newly added feature. `wsimport` provides proxy classes to interact with the target system.

proach to determine the data-flow between method invocations and to choose parameter and return variables for the setup specification.

This means that information on data dependencies between operations, as deduced by `StrawBerry`, are used to construct an alphabet of parameterized actions. This allows for carrying enough information so that the mapper can translate abstract alphabet symbols into concrete actions outfitted with live data values and manage return values of invocations.

To illustrate how an automated learning setup can be instantiated with the help of the generated setup descriptor (an activity represented as *Test-driver and mapper instantiation* in Figure 5), it is helpful to recall which concerns have to be addressed:

- Means for instrumentation of the SUL have to be provided, e.g., by means of a proxy that is accessible by the test-driver.
- An alphabet for the learner has to be constructed, as well as a mapping between the abstract alphabet and concrete system actions. This is where the dependency information provided by Strawberry is essential.
- Facilities for handling communicated data-values have to be present and configured to account data-flow between operations.

In the following we will discuss these points in more detail:

5.1 Instrumentation

Within a setup for active automata learning, the instrumentation layer is responsible for injecting system stimuli and gathering the target system's output for every invocation. For WSDL services, injecting system stimuli can be done in a straightforward way, e.g., by using automatically generated proxy classes that expose system functionalities while hiding the specifics of operating the target system through networked messages. For the experiments discussed in this paper, proxy classes for the remote system are generated by the `wsimport` [18] utility, which can serve as instrumentation layer for the test driver (denoted as *WS import* activity in Figure 5).

5.2 Determining an Alphabet and Mapper Configuration

The interface description is a natural source for the alphabet employed for the learning process, as every message defined in the WSDL description usually has a direct mapping to system features intended for remote consumption. It appears most sensible to choose the names of the defined WSDL messages as abstract alphabet symbols for the learner, which the test-driver concretizes into actual operation invocations of the generated proxy classes. As such the mapping between the abstract learning alphabet and concrete system input is one from operation names to actual invocations of the corresponding operation.

For parameterized operations, abstract alphabet symbols also have to include information for the mapper on how to retrieve values from the data value context to enable actual invocation. Thus the abstract symbols for parameterized operation calls will contain references to this context in form of instructions on how to retrieve data values from it.

To populate the data value context, data values returned by the SUL will be stored as named variables. Thus the abstract symbols also have to contain information on the name of the variable the return value is assigned to. For each stored value the abstract output symbol forwarded to the learner will simply be the variable name in which the return value was stored, abstracting from the actual content of the return message that the system under test produced. A similar approach to abstraction is taken for error messages: if the SUL produces an elaborate error message, the output returned to the learner usually will be a generic "error" symbol, abstracting from all the details related to this error instance. No data value will be stored in this case.

5.3 Storing and Accessing Data-Values

When concretizing learning queries into actual system input, fitting data values have to be inserted into parameterized system messages. Thus the system driver has to be able to store received data values and generate concrete system input by resorting to these stored values.

To accommodate data values, the data value context is realized as an embedded JavaScript environment. The reason for choosing a JavaScript environment over, e.g., a map of variable names and variable values, lies in the ability of a scripted context to access stored data with utmost flexibility. A scripted data value context is, e.g., able to access fields of complex data structures and provide those as parameter values.

Not every parameter can be filled with data values that are results of preceding system invocations. One notable example for this are login credentials, which have to be known beforehand. Such values have to be included in the setup specification and are copied into the data value context.

6 Application to the Example and Discussion

In the following, we will apply the presented approach to the running example.

Figure 6 shows an excerpt of the setup descriptor created by `StrawBerry` as a result of the interface analysis. The `serviceurl` declaration in line 2 provides an URL to the SUL, which can be directly used as an input for `wsimport`. Predetermined values (credentials in this case) are provided in lines 3 to 6 and are used to populate the instance pool.

The remainder of the specification file defines a sequence of symbols. Each `symbol` includes a sequence of `parameter` declarations, which refer to named variables in the data value context. It can be seen that the `symbol` declarations include information on parameters and on the variables where return values are stored. Parameter values stored in the data value context are addressed by named keys that are specified by the `alternative` environment. The reason for having `alternative` declarations is that parameters may have several potential data sources. For example, the second parameter of the symbol `addProductToShoppingCart` may take data values from the variables `productArray` and `shoppingCart`. Each alternative induces the instantiation of additional abstract symbols, meaning that for the presented example the learning alphabet has in fact two `addProductToShoppingCart` symbols, one referring to `productArray` as parameter value, the other referring to `shoppingCart`.

```
<learnsetup>
 <serviceurl>http://vulpis.cs.tu-dortmund.de:9000/ecommerceservice?wsdl
    </serviceurl>
 <provided>
  <object name="username" type="string">username</object>
  <object name="password" type="string">password</object>
 </provided>
 <symbols>
  <symbol name="openSession">
   <parameters>
    <parameter>
     <alternative>username</alternative>
    </parameter>
    <parameter>
     <alternative>password</alternative>
    </parameter>
   </parameters>
   <return>session</return>
  </symbol>
  ...
  <symbol name="getAvailableProducts">
   <parameters />
   <return>productArray</return>
  </symbol>
  ...
  <symbol name="addProductToShoppingCart">
   <parameters>
    <parameter>
     <alternative>session</alternative>
    </parameter>
    <parameter>
     <alternative selector="elementOf" field="item">productArray
        </alternative>
     <alternative selector="elementOf" field="items">shoppingCart
        </alternative>
    </parameter>
   </parameters>
   <return>session</return>
  </symbol>
 </symbols>
</learnsetup>
```

Fig. 6. Excerpt of the setup descriptor for `LearnLib` generated by `StrawBerry` by means of syntactical analysis

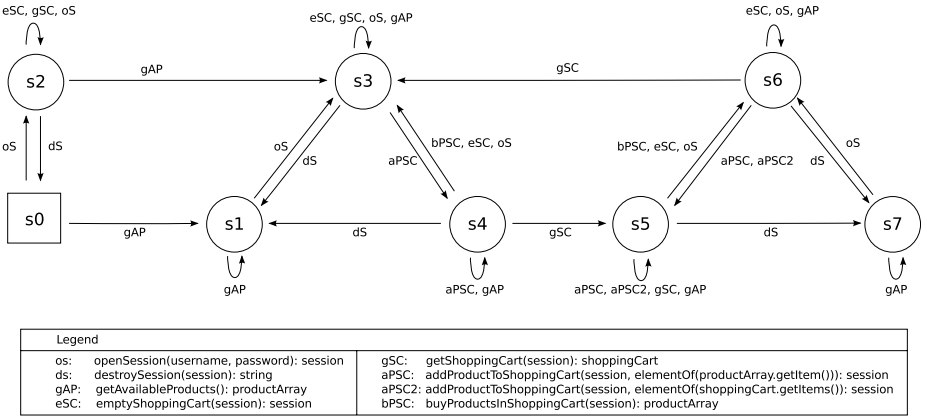

Fig. 7. Model created by `LearnLib` using the setup description created by `StrawBerry`. The edge labels are abbreviated for improved readability.

The parameters of the symbol `addProduct- ToShoppingCart` illustrate why realizing the data value context as scriptable environment is advantageous: the `alternative` declaration in line 31 of Figure 6 includes attributes that specify how the data value for the corresponding parameter has to be extracted from the context. Instead of directly filling in the parameter value with the complete data structure that is pointed by the variable `productArray`, only the `field` "item" of this data structure should be considered. However, the field "item" references a set of products and not a single product. Thus, the `selector` "elementOf" is specified as well. From this information the JavaScript expression `elementOf(productArray.getItem())` is derived and evaluated on the data value context at run time, where the function `elementOf()` is predefined and simply returns the first value of any provided collection.

The result of learning a behavioral model with this setup specification is shown in Figure 7. Please note that this figure presents a view onto the learned Mealy machine that omits error transitions, only showing actions that do not raise an exception during execution. The impact of the experimental semantical analysis is already apparent from the fact that this model contains more states than those created by `StrawBerry` by means of syntactic analysis and test runs, with the effect that all the three properties mentioned in Section 2 are correctly revealed:

- When no products have previously been added to the shopping cart, the operation to purchase products does not conclude successfully: the purchase action only succeeds in states s4 and s5, which can only be reached when adding at least one product to the shopping cart.
- As long as a session is open, it is possible to empty its associated shopping cart: the action to empty the shopping cart succeeds in all states except the states s0, s1 and s7, where the session either has not been openend yet or was subsequently destroyed.

– After a purchase operation it is not possible to immediately trigger another purchase. Instead, it is necessary to put another item into the shopping cart, which implies that the purchase operation does clear the shopping cart. This can be witnessed at the s4/s3 and s5/s6 transitions.

Apart from these facets even more subtle behavioral aspects are captured. For example, once a non-empty shopping cart is retrieved, its contents can be added to another session's shopping cart. This means that the data structure representing products in a shopping cart is not bound to session instances, which is another implementation detail influencing how the service can be operated that is not explicitly contained in the service's interface description.

7 Related Work

Inferring formal properties of software components has been a major research interest for the past decade. Most available approaches fall into one of two classes. One class generates extrinsic properties (e.g., invariants). The other class generates intrinsic properties, e.g., models describing the actual behavior of components. In both classes active and passive approaches, as well as black-box and white-box variants can be found. While `StrawBerry` falls into the first category, `LearnLib` is of the second kind.

The class of methods for generating properties can be further subdivided into methods that "mine" statistical models and methods that generate invariants. In the class of methods that generate statistical models, the approaches described in [25,24] mine Java code to infer sequences of method calls. These sequences are then used to produce object usage patterns and operational preconditions, respectively, that serve to detect object usage violations in the code. `StrawBerry` shares with [24] the way an object usage pattern is represented, i.e., as a set of temporal dependencies between method calls (e.g., $m < n$ means "calls to m precede calls to n").

The work of [19] presents a passive method for the automated generation of specifications of legal method call sequences on multiple related objects from from method traces of Java programs, [9] extends this method by active testing. As for `StrawBerry`, tests are used to refine the invariants that have been generated inductively from the provided information. However, in contrast to `StrawBerry`, none of these approaches focuses on data-flow invariants explicitly. A tool that infers invariants from data is Daikon [10].

In the class of methods that generate intrinsic properties, especially automata learning has been used to generate behavioral models of systems. Active learning, as implemented in `LearnLib`, has been used to infer behavioral models of CTI systems as early as 2002 [12,13]. It has since then been applied in a number of real-life case studies (e.g., [21,20]). In these case studies, however, data has never been treated explicitly but was rather hidden from the learning algorithm. In [22], systems with data parameters are considered. However, this work does not consider relations between different parameters. Recently, automata learning has been extended to deal with data parameters and data dependencies explicitly by means of hand-crafted mappers [14,1]. Our approach is unique in generating mappers automatically.

There are only few approaches that combine inference of behavioral models and invariants on data-flow. The authors of [5] present an approach for inferring state machines (by means of active learning) for systems with parameterized inputs. They first infer a behavioral model for a finite data domain, and afterwards abstract this model to a symbolic version, encoding extrapolated invariants on data parameters as guarded transitions.

The authors of [16,15] demonstrate how behavioral models can be created with passive learning from observations gathered by means of monitoring. In addition, this approach tries to capture the behavioral influence of data values by applying an invariance detector [10]. This approach, however, is subject to the issue of all passive approaches: they are limited to the (possibly small) set of observed executions. If a piece of code or part of the application is not executed, it will not be considered in the generated model.

The work described in [11] (i.e., the SPY approach) aims at inferring a behavioral specification (in this case: graph transformation rules) of Java classes that behave as data containers components by first observing their run-time behavior on a small concrete data domain and then constructing the transformation rules inductively from the observations.

It is common to all these approaches that they work on a large basis of concrete information that by induction is condensed into symbolic behavioral models. Invariants on data values are obtained in a post-processing step after construction of behavioral models.

In [2] an approach is presented that generates behavioral interface specifications for Java classes by means of predicate abstraction and active learning. Here, predicate abstraction is used to generate an abstract version of the considered class. Afterwards a minimal interface for this abstract version in obtained by active learning. This is a white-box scenario, and learning is used only to circumvent more expensive ways of computing the minimal interface.

Our approach, in contrast, provides a solution for the black-box scenario. Similarly, however, we use `StrawBerry` to compute an interface alphabet, and mapper, which in combination work as an abstraction, and infer a model at the level of this abstraction, using `LearnLib`.

8 Conclusions and Perspectives

We have presented a method and tool to *fully automatically* infer dataflow-sensitive behavioral models of black-box systems based on interface descriptions in WSDL by combining `StrawBerry`, a tool for syntactical analysis of the interface descriptions and the `LearnLib`, a flexible active automata learning framework. This combination allows us to overcome a central bottleneck, the manual construction of the mapper required for the learning tool to bridge between the model level and the concrete execution level.

Our method has been illustrated in detail along a concrete shop application example. The results are promising, but further case studies are required to fully explore the application profile of the approach. Scalability is certainly an issue here, and it has to be seen how stable the approach is concerning varying versions of WSDL-based interface specifications. Particularly interesting is here to investigate how our approach

may profit from extra information provided e.g. through semantic annotations, a point explicitly addressed also in the CONNECT context. There, full automation is not sufficient as CONNECT's support is meant to happen fully online. Finally, we are currently working on an extension of our technology to generate even more expressive models in terms of register automata [8]. These models are designed to make the currently only implicitly modeled dataflow information explicit by introducing transitions with explicit conditions and assignments.

References

1. Aarts, F., Jonsson, B., Uijen, J.: Generating Models of Infinite-State Communication Protocols Using Regular Inference with Abstraction. In: Petrenko, A., Simão, A., Maldonado, J.C. (eds.) ICTSS 2010. LNCS, vol. 6435, pp. 188–204. Springer, Heidelberg (2010)
2. Alur, R., Cerný, P., Madhusudan, P., Nam, W.: Synthesis of interface specifications for Java classes. In: POPL, pp. 98–109 (2005)
3. Angluin, D.: Learning regular sets from queries and counterexamples. Inf. Comput. 75, 87–106 (1987)
4. Bennaceur, A., Blair, G., Chauvel, F., Gang, H., Georgantas, N., Grace, P., Howar, F., Inverardi, P., Issarny, V., Paolucci, M., Pathak, A., Spalazzese, R., Steffen, B., Souville, B.: Towards an Architecture for Runtime Interoperability. In: Margaria, T., Steffen, B. (eds.) ISoLA 2010, Part II. LNCS, vol. 6416, pp. 206–220. Springer, Heidelberg (2010)
5. Berg, T., Jonsson, B., Raffelt, H.: Regular Inference for State Machines Using Domains with Equality Tests. In: Fiadeiro, J.L., Inverardi, P. (eds.) FASE 2008. LNCS, vol. 4961, pp. 317–331. Springer, Heidelberg (2008)
6. Bertolino, A., Inverardi, P., Pelliccione, P., Tivoli, M.: Automatic synthesis of behavior protocols for composable web-services. In: ESEC/SIGSOFT FSE, pp. 141–150. ACM (2009)
7. Bertolino, A., Inverardi, P., Pelliccione, P., Tivoli, M.: Automatic synthesis of behavior protocols for composable web-services. In: Proceedings of The 7th Joint Meeting of the European Software Engineering Conference (ESEC) and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE), pp. 141–150 (August 2009)
8. Cassel, S., Howar, F., Jonsson, B., Merten, M., Steffen, B.: A Succinct Canonical Register Automaton Model. In: Bultan, T., Hsiung, P.-A. (eds.) ATVA 2011. LNCS, vol. 6996, pp. 366–380. Springer, Heidelberg (2011)
9. Dallmeier, V., Knopp, N., Mallon, C., Hack, S., Zeller, A.: Generating test cases for specification mining. In: Proceedings of ISSTA 2010, pp. 85–96. ACM, New York (2010)
10. Ernst, M.D., Perkins, J.H., Guo, P.J., McCamant, S., Pacheco, C., Tschantz, M.S., Xiao, C.: The Daikon system for dynamic detection of likely invariants. Sci. Comput. Programming 69(1-3), 35–45 (2007)
11. Ghezzi, C., Mocci, A., Monga, M.: Synthesizing Intentional Behavior Models by Graph Transformation. In: ICSE 2009, Vancouver, Canada (2009)
12. Hagerer, A., Hungar, H., Niese, O., Steffen, B.: Model Generation by Moderated Regular Extrapolation. In: Kutsche, R.-D., Weber, H. (eds.) FASE 2002. LNCS, vol. 2306, pp. 80–95. Springer, Heidelberg (2002)
13. Hungar, H., Margaria, T., Steffen, B.: Test-based model generation for legacy systems. In: Proceedings of the International Test Conference, ITC 2003, September 30-October 2, vol. 1, pp. 971–980 (2003)
14. Jonsson, B.: Learning of Automata Models Extended with Data. In: Bernardo, M., Issarny, V. (eds.) SFM 2011. LNCS, vol. 6659, pp. 327–349. Springer, Heidelberg (2011)

15. Lorenzoli, D., Mariani, L., Pezzè, M.: Automatic Generation of Software Behavioral Models. In: ICSE 2008, pp. 501–510. ACM, NY (2008)
16. Mariani, L., Pezzè, M.: Dynamic Detection of COTS Component Incompatibility. IEEE Software 24(5), 76–85 (2007)
17. Niese, O.: An Integrated Approach to Testing Complex Systems. PhD thesis, University of Dortmund, Germany (2003)
18. Oracle.com. JAX-WS RI 2.1.1 – wsimport, http://download.oracle.com/javase/6/docs/technotes/tools/share/wsimport.html (2011) (online; accessed September 13, 2011)
19. Pradel, M., Gross, T.: Automatic generation of object usage specifications from large method traces. In: Proceedings of ASE 2009, pp. 371–382 (November 2009)
20. Raffelt, H., Margaria, T., Steffen, B., Merten, M.: Hybrid test of web applications with webtest. In: Proceedings of TAV-WEB 2008, pp. 1–7. ACM, New York (2008)
21. Raffelt, H., Steffen, B., Berg, T., Margaria, T.: Learnlib: a framework for extrapolating behavioral models. Int. J. Softw. Tools Technol. Transf. 11, 393–407 (2009)
22. Shahbaz, M., Li, K., Groz, R.: Learning Parameterized State Machine Model for Integration Testing, vol. 2, pp. 755–760. IEEE Computer Society, Washington, DC (2007)
23. Shahbaz, M., Shashidhar, K.C., Eschbach, R.: Iterative refinement of specification for component based embedded systems. In: Proceedings of ISSTA 2011, pp. 276–286. ACM, New York (2011)
24. Wasylkowski, A., Zeller, A.: Mining Operational Preconditions (Tech. Rep.), http://www.st.cs.uni-saarland.de/models/papers/wasylkowski-2008-preconditions.pdf
25. Wasylkowski, A., Zeller, A., Lindig, C.: Detecting Object Usage Anomalies. In: ESEC-FSE 2007, pp. 35–44. ACM (2007)