# Automated Learning Setups in Automata Learning*

Maik Merten[1], Malte Isberner[1], Falk Howar[1], Bernhard Steffen[1], and Tiziana Margaria[2]

[1] Technical University Dortmund, Chair for Programming Systems, Dortmund, D-44227, Germany
{maik.merten|malte.isberner|falk.howar|steffen}@cs.tu-dortmund.de
[2] University Potsdam, Chair for Service and Software Engineering, Potsdam, D-14482, Germany
margaria@cs.uni-potsdam.de

**Abstract** Test drivers are an essential part of any practical active automata learning setup. These components to accomplish the translation of abstract learning queries into concrete system invocations while managing runtime data values in the process. In current practice test drivers typically are created manually for every single system to be learned. This, however, can be a very time-consuming and thus expensive task, making it desirable to find general solutions that can be reused.
This paper discusses how test drivers can be created for LearnLib, a flexible automata learning framework. Starting with the construction of application-specific test drivers by hand, we will discuss how a generic test driver can be employed by means of configuration. This configuration is created manually or (semi-)automatically by analysis of the target system's interface.

## 1 Introduction

In recent years, automata learning has been employed to create formal models of real-life systems, such as electronic passports [1], telephony systems [5,7], web applications [14,15], communication protocol entities [3], and malicious networked agents [4]. The wide scope of application areas gives testimony on the universality of the automata learning approach.

However, challenges remain regarding the construction of application-specific learning setups. A major obstacle for widespread deployment of active automata learning is the effort needed to design and implement application-fit learning setups. This involves determining a suitable form of abstraction and finding ways to manage concrete runtime data that influences the behavior of the target system. In [16], the combined effort for constructing an application-specific abstraction

* This work was partially supported by the European Union FET Project CONNECT: Emergent Connectors for Eternal Software Intensive Networked Systems (http://connect-forever.eu/).

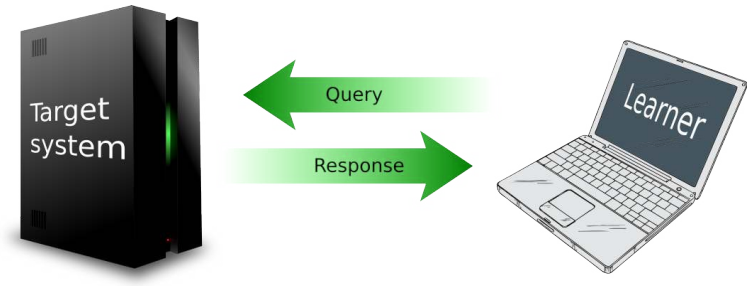


**Figure 1.** High-level overview on an active learning setup

and a test driver is estimated to have consumed approximately 27% of the total effort of analyzing an embedded system from the application area of automotive systems.

Learning aims at inferring an abstract model of the SUL. While the chosen abstraction has influence on the expressiveness of the final learned model, dealing with concrete runtime data is an immediate concern when interacting with reactive systems where communication often is dependent on concrete data values previously transferred. For example, a system guarded by an authorization system may transport a security token to the client on login, which then has to be included in any interaction with protected system areas.

In order to support the full communication of the learner with the SUL, the learning setup has to translate abstract learning queries into concrete requests to the target system. These concrete requests may have to be outfitted with data values. In automata learning, the building block facilitating the translation is a so-called mapper [8]. In this paper we show how to manually create test drivers that include mapper functionality, and discuss how a reconfigurable and reusable test driver can be set up by means of interface analysis.

## 2 Active automata learning

In active automata learning, models of a target system—here denoted as *SUL* (*System Under Learning*)—are created by active interaction and by reasoning on the observed output behavior. This is done by constructing *queries*, which are sequences of *input symbols* from an alphabet that represents actions executable on the SUL, and answering these queries by means of actual execution. A high-level overview of the structure of an active automata learning setup is provided in Figure 1.

There exists a variety of different active learning algorithms that interrogate the SUL in the described fashion. A selection of algorithms, complete with corresponding infrastructure, is provided with LearnLib [13,11], a versatile automata learning framework available free of charge at `http://learnlib.de`.

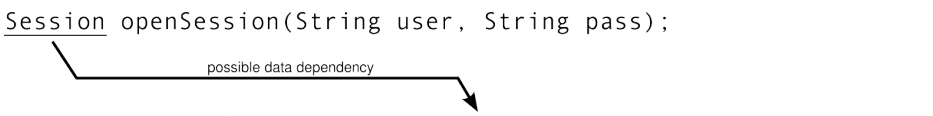


**Figure 2.** A possible data dependency between method calls

## 3 A running example

In this paper, we will discuss the construction of test drivers along the example of a fictional e-commerce application where users can log in, retrieve a list of products, add products to their shopping cart, and finally buy its contents. This example has been implemented as web service exposing a WSDL interface and thus offers a standardized and networked way of interaction. Following methods are exposed:

- `openSession` expects user credentials and returns an authentication token. Conversely, `destroySession` invalidates a specified session and the associated shopping cart.
- `getAvailableProducts` returns a list of available products.
- `addProductToShoppingCart` expects an authentication token identifying a user session and adds a provided product to the associated shopping cart. Conversely, the `emptyShoppingCart` primitive empties the shopping cart of a specified session. The method `getShoppingCart` returns a representation of the session's shopping cart, with references to all products it contains.
- `buyProductsInShoppingCart` will purchase the contents of the shopping cart associated with the provided session.

When interacting with this example system, the following challenges have to be addressed, and we will refer to these challenges when demonstrating ways to establish application-fit learning setups:

*Data dependencies:* To be able to learn this system, the learning setup needs to deal with the data dependencies between methods. For instance, most actions require a valid authentication token which is provided by the `openSession` primitive. However, this method again is dependent on data values, namely valid login credentials, which have to be provided beforehand. This situation is illustrated in Figure 2.

*Dependencies on substructures:* Merely filling in parameters with runtime values is not sufficient to interact with this system. For instance, the `addProductToShoppingCart` method expects a single product to be provided. The `getAvailableProducts` method provides a collection of fitting data values, but the returned data structure cannot be directly used as a parameter value for `addProductToShoppingCart`, that expects only a single data value, as illustrated

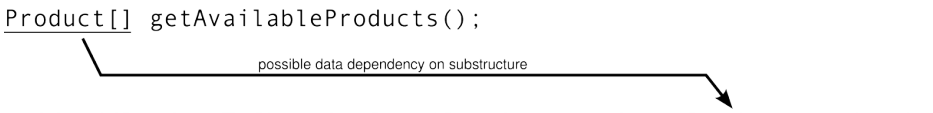


**Figure 3.** A data dependency involving a singular value out of a collection of values

in Figure 3. Determining a fitting valuation for this parameter requires a basic understanding of the application's data structures, accompanied by means to execute basic operations on these data structure such as, e.g., isolating single data values out of a collection of values.

These two forms of dependencies imply a required, but not sufficient order on method calls. For instance, the method `buyProductsInShoppingCart` needs a valid session identifier to conclude a purchasing transaction, implying that the method `openSession` needs to be called beforehand. However, this alone is not sufficient, as empty shopping carts cannot be purchased, which is a behavioral aspect arising from the stateful nature of the system that cannot be determined by data dependency analysis alone. Active automata learning, however, is able to fill in these state-dependent behavioral traits.

In the following we discuss an architecture for test drivers that enables dealing with these challenges.

## 4 Test drivers and mappers in active automata learning

In most real-life automata learning applications, learning alphabets impose an abstraction on the actual interaction with the SUL. For instance, a sequence of several concrete input symbols of the SUL may be combined into one single abstract symbol that represents a single use case.

Consequently, as the active automata learning procedure has to procure the production of observable system output, these abstract learning alphabets have to be translated into concrete system alphabets, i.e., alphabets composed of inputs the target system can process. Conversely, the concrete system output has to be translated into abstract output symbols that fit the intended model structure.

In practice, this two-way translation process can be handled by a test driver, which can be integrated seamlessly into LearnLib's modular framework. Figure 4 shows a component-wise view onto such a test driver, embedded within a learning setup. In this figure, the following core components are visible:

- A *mapper* is responsible for bridging the gap between abstract and concrete alphabets, i.e., the mapper is responsible for the translation of learning queries composed of abstract input symbols into queries composed of concrete system inputs. For parameterized symbols, the mapper also determines

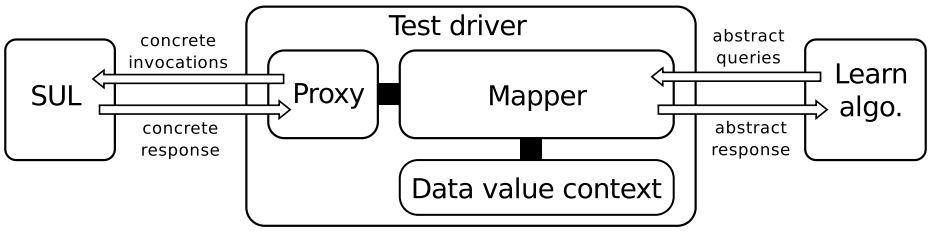


**Figure 4.** General architecture of a test driver for active automata learning

fitting parameter valuations and inserts these data values accordingly. Referring to the running example, the mapper concretely invokes the `openSession` action on the SUL with fitting credentials whenever an abstract symbol such as "login" is encountered. The concrete return value of this method, which differs for every invocation, needs to be abstracted to gather reproducible observations. This can be done, e.g., by emitting an output symbol that merely denotes invocation success or failure.

– The *data value context* supports the mapper whenever parameterized actions have to be translated. Many interactions with SULs require parameter values, e.g., login procedures need preset credentials that do not change during learning, while subsequent actions may require an authentication token determined at runtime. The data value context manages such concrete values from the application's data value domain, which is a prerequisite for overcoming both data-related challenges outlined in Section 3. Data values are fetched and updated according to requests issued by the mapper component during concretization and abstraction steps. During concretization data values are fetched from the data value context and used by the mapper for parameterized invocations. Consequently, when abstracting from concrete return values, the mapper will generate a fitting abstract output symbol, but will also issue a request to the data value context to store the concrete data value for future reference. In the running example, the `openSession` action returns an authentication token whose concrete value needs to be stored for methods such as `addProductToShoppingCart`.
– The *proxy* is a component that directly interfaces with the SUL, maintains a connection and thus serves as the funnel to direct learning queries to the target system. Responses of the target system are collected by the proxy and transferred into concrete output symbols subsequently processed by the mapper component. The main purpose of the proxy thus is to facilitate interaction with the SUL by means of a unified invocation mechanism (e.g., simple Java methods), abstracting from the underlying invocation technology (such as, e.g, SOAP, RMI or CORBA). For systems with an interface description in a standardized format such as WSDL, fitting proxy objects can often be generated fully automatically by employing connector generation tools for that standardized format. The example application falls within this category: it exposes a WSDL interface that can be converted into in-

vocable code by a tool that emits a Java class encapsulating the remote invocation mechanics.

Providing these three components can be a major bottleneck when preparing real-life learning scenarios. This effort includes thoughtful construction of the involved abstraction layers and implementation of the according translation mechanisms, i.e., the construction of a fitting mapper.

## 5 Manual construction of test drivers and setups

In LearnLib, any components that answer learning queries need to implement the `MembershipOracle` interface. A test driver implemented according to this interface possesses one single method `processQuery` providing system output in response to system input, i.e., it generates output for learning queries.

Figure 5 shows a manually created test-driver for the example system described in Section 3. For reasons of simplicity, not all actions available on the target system are implemented in this test driver. Regarding the core components of the test driver, the following implementations can be observed:

- The *mapper* is implemented using hardwired abstraction and concretization steps, e.g., by invoking the `openSession` method of the target system when the abstract input symbol "login" is encountered. In the code example, the mapping between abstract symbols and concrete invocations is realized employing simple `if` statements (lines 20 to 26). In a similarly coarse fashion system output is abstracted as "ok" if no error was signaled, as "error" otherwise (lines 26 and 30 respectively). In effect this means that both the abstract input alphabet ("login", "getProducts", and "addProduct") and the abstract output alphabet ("ok" and "error") are fixed, as is the mapping from the abstract input alphabet to the concrete system invocations (methods `openSession`, `getAvailableProducts` and `addProductToShoppingCart`). Note that this particular test driver does not support any additional symbols: for instance, to actually conclude a purchase, it would have to be extended accordingly.
- The variables `session` and `products` in the `processQuery` method (lines 11 and 12) are used as a *data-value context* to resolve *data dependencies*. The former is employed to store the invocation result of the `openSession` method, the latter stores a collection of product information returned by `getAvailableProducts`. The credentials for the `openSession` method (line 21) are hardcoded strings which were determined beforehand.
  As described in Section 3 on the challenge of *dependencies on substructures*, direct use of runtime data as parameter valuations is not always sufficient. This is visible in line 25, where the `addProductToShoppingCart` action is invoked. There, the second parameter is instantiated using the `products` variable. However, instead of passing the whole collection of products as parameter, a single value is selected (in this case always the first element). This constitutes an operation upon a data structure previously returned by

```
1  public class TestDriver implements MembershipOracle {
2
3    private ShopSystem system = new ShopSystem();
4
5    @Override
6    public Word processQuery(Word query) throws LearningException {
7      // output word collecting system reaction
8      Word output = new WordImpl();
9
10     // variable to store authentication token
11     Session session;
12     Product[] products;
13
14     for (int i = 0; i < query.size(); ++i) {
15       // retrieve current symbol from query
16       Symbol inputsym = query.getSymbolByIndex(i);
17
18       try {
19         // act on system according to abstract symbol
20         if (inputsym.toString().equals("login")) {
21           session = system.openSession("username", "password");
22         } else if (inputsym.toString().equals("getProducts")) {
23           products = system.getAvailableProducts();
24         } else if (inputsym.toString().equals("addProduct")) {
25           system.addProductToShoppingCart(session, products[0]);
26         }
27
28         // no error
29         output.addSymbol(new SymbolImpl("ok"));
30
31       } catch (Exception e) {
32         // error signalled via system exception
33         output.addSymbol(new SymbolImpl("error"));
34       }
35     }
36
37     return output;
38   }
39 }
```

**Figure 5.** A manually created test-driver

the application, which involves a basic understanding of the organization of the affected data structure.
- The *proxy* in this example is provided in the form of the `system` variable (line 3), which contains a reference to an object directly exposing the SUL's methods, e.g., an object generated from the system's WSDL interface description. This object encapsulates interacting with the SUL by means of network messages, shielding the test driver developer from interaction details such as maintaining a network connection and assembling, e.g., SOAP (Simple Object Access Protocol) messages. Thus the proxy object enables interaction with the target system by means of simple method invocations, as is done in lines 21, 23 and 25.

Clearly, hand-tailoring fitting test drivers for more complex systems can quickly become a bothersome, time-consuming (and thus expensive) task. To make matters worse, such test drivers are not reusable for any other system than for the original SUL and offer only limited flexibility even when considering a single system, because each adaption necessitates code changes.

The following sections will discuss how the setup effort can be dramatically reduced, to the point of approaching fully automated construction and execution of learning setups.

## 6 Constructing learning setups by interface analysis

Key to automated instantiation of learning setups is the development of flexible, configurable test drivers. Such a test driver was developed for LearnLib, which can operate on a wide range of systems [12]. It is structured as follows:

- The *mapper* translates abstract input symbols into concrete Java method invocations of the proxy. The return values are stored in the data value context as named variables. Abstract output symbols named after these variables are returned on success. If, e.g., the proxy signals a system exception, an abstract error symbol is emitted instead. In contrast to the manually constructed test driver discussed in Section 5 the abstraction function is not hard coded, but configurable.
- As *data value context* a JavaScript context is employed. It can not only store named variables to resolve *data dependencies*, and also allows the execution of data retrieval operations, such as isolating single data values from complex data structures such as collections to resolve the challenge of *dependencies on substructures*. The data value context is also employed to store predefined data values such as login credentials.
- The *proxy* is a Java object upon which methods are invoked employing the Java reflection API. While in Section 5 the proxy object was hardcoded in the test driver, the configurable test driver is designed to generate a proxy object at runtime from an interface description and subsequently use it for system invocation. This is currently implemented for WSDL, employing the `wsimport` utility, so it suffices to provide only an URL to the interface description.

When employing such a test-driver, fitting configurations must be determined for the mapper, the proxy generation and the data value context. This boils down to the questions of how to construct an alphabet, how to locate to the SUL's interface description, and how to manage live data values necessary to drive interaction with the target system. The questions can be answered by interface analysis, as illustrated in the following.

### 6.1 Constructing the alphabet

Most APIs are structured with some sense of abstraction in mind. In fact, a major purpose of well-designed APIs is the abstraction from the underlying implementation details, offering application features in a structured and meaningful way.

When documenting how to interact with a target system, the abstraction level imposed by the design of the system's API is a natural abstraction level of the model that is to be created for documentation purposes. Thus, an alphabet can be constructed in a straightforward way:

- Every *method* in the API can be translated into an abstract symbol of the learning alphabet. The runtime semantics of these abstract learning symbols is the concrete invocation of the corresponding method exposed by the API.
- *Parameters* of API methods are handled by parameterizing the abstract learning symbols of parameterized interface methods. At runtime, fitting valuations have to be retrieved from the data value context and included in the concrete system invocations. Data values can be stored in named variables in the data value context. Parameters in abstract learning symbols subsequently refer to these variable names.
- *Return values* can be abstracted according to the return type, i.e., the abstract output symbol merely denotes that a data value of a specific type has been returned. The concrete live data values are delivered to the data value context and stored in variables named after the corresponding return types. In effect this means that only one data value per data type can be stored, a limitation which precludes, e.g., the possibility of invoking actions that require two distinct values of the same type. For systems that employ a single data type to encode data values with distinct purposes (e.g., if all data values are encoded as character strings) this limitation can severely restrict the ability to interact with the SUL, necessitating a refined approach for output abstraction. As demonstrated for learning Register Automata, it is possible to determine the exact set of data values that have to be memorized [6].

In case of standard Java interfaces, the necessary analysis steps can easily be done using the class reflection scheme that is part of the Java platform. Cross-platform interface description formats can usually be parsed by specialized tools in a comparable fashion.

### 6.2 Interfacing with the target system

The configurable test driver includes a component to generate a proxy object from interface descriptions, which is currently implemented for WSDL interface descriptions. The `wsimport` tool employed generates a Java class that exposes the methods defined in the interface description and handles all networked communication with the SUL, abstracting from the underlying protocol details. Thus, from the perspective of the test driver, proxy objects generated by `wsimport` are merely normal Java objects, with methods that can be invoked dynamically at runtime by the Java reflection mechanism. Apart from WSDL web services (such as the discussed example system) this approach is also feasible for other remote invocation technologies, such as CORBA, for which similar code generation tools exist.

### 6.3 Managing live data values

Method calls in interfaces often depend on parameters that are instantiated with runtime data. For example, a method may produce data values that are consumed by a consecutive method call. This is easy to witness in the example e-commerce scenario of Section 3, where one API method produces an authentication token that has to be provided by other methods of the system (a situation illustrated in Figure 2). This sort of data dependency must be satisfied with live data values determined at runtime. To be able to solve this problem with no or little manual intervention, such data dependencies must be determined automatically. In the following, a solution is sketched:

- In case of interfaces with strongly typed data, data dependencies can only exist in alignment with the type concept, i.e., a value returned by one method can only be provided as input parameter for another method if the return value type equals (or is a subtype of) the parameter type. Consequently, no data dependencies have to be assumed outside of the type hierarchy. In the example sketched above, one method may produce a sequence of values, each typed as "Product", which can subsequently be consumed by another method. Thus the former is a potential producer of viable data values for the latter.
  This type of analysis is bound to be impractical if the interface is specified over a depleted type system. For instance, many web services encode all or most data values as simple character strings. From the perspective of the type system thus any data values could apply "anywhere", devoid of any semantic meaning.
- If no data type concept is present (or if a depleted type concept is employed as described above), the syntactic analysis over data types can be replaced or augmented by a testing phase in which active interaction with the target system determines which return values are fitting input for parameters of subsequent method calls. This, in effect, means that static analysis of a strong type system is replaced by a training phase to determine a type system

regarding interoperability of method calls. A tool for performing this kind of analysis on WSDL interfaces is `StrawBerry` [2].

Once the relation between methods and involved data types has been determined, the data flow induced by data dependencies can easily be realized by allocating one variable in the data value context per data type. Parameter values can be retrieved according to the parameter type and return values can be stored according to the returned data type. This scheme can be implemented with a data value context that in essence is a map containing data values associated with keys corresponding to the involved data types.

It is easy to see how simple data dependencies over single data values can be handled in this fashion.

However, the *dependencies on substructures* challenge described in Section 3 eludes this simple treatment as shown in Figure 3, where one method provides a sequence of values, while the other method consumes single values. This means that merely providing the returned sequence as parameter value is not an option. While it is possible to detect this situation during in-depth type analysis, a conventional map data structure is not a fitting implementation for the data value context, as simple operations such as isolation of single data values out of data value sequences are needed. The same problem occurs when only a single attribute of a complex data type has to be provided as an argument to another method call.

For this reason the data value context of the reconfigurable test driver employs a scriptable JavaScript context that can execute arbitrary program statements on stored data values, such as, e.g., "`elementof(collection)`", which retrieves one single data value out of a collection, and also supports the common dot notation for accessing attributes and methods of complex types. These statements are included in the abstract parameterized learning symbols and are evaluated as provided by the mapper component that inspects symbols of the abstract learning alphabet as part of the mapping process.

### 6.4 Employing semantic analysis

In Section 6.3, type analysis was employed to determine data-flow between invocations of the SUL. For cases where the type system of the interface description was nondescript or even missing, a testing phase was proposed to experimentally determine data dependencies between method invocations.

Any such testing procedure, however, may yield unsatisfactory results, depending on the complexity of inter-method data dependencies and the employed coverage criteria used during the testing phase. Thus data dependencies may be missed, causing the construction of incomplete system models in the subsequent active learning phase.

Due to the limitations of pure syntactical interface analysis, which can detect false data dependencies if generic data types are used as parameters and return types, and test-based analysis of data dependencies, which can miss data dependencies if testing is not thorough enough, an alternative approach is desirable.

One such approach is based on explicitly specifying the semantical concepts of parameters and return values in a way that is independent of the type system. For WSDL, an extension called *Semantic Annotations for WSDL* (SAWSDL) has been proposed [17]. Using SAWSDL, data occurring in the interface description—not only on the level of formal parameters, but also for attributes of complex types—can be annotated with a reference to a concept in an Ontology. A common example is distinguishing the semantic concepts of the username and password parameters of a login operation, which usually are both strings, even in case of depleted type systems. Using an OWL reasoner like Pellet,[3] also more complex relations like subclassing and inferring class membership can be realized.

This approach crucially relies on semantic annotations (and a corresponding ontology) being available, an assumption which is false for most third-party web services. Despite allowing the most fine-grained inference of data dependencies, we will therefore not detail this approach here any further, as its applicability to real-world use cases is limited.

## 7 The setup interchange format

The result of the analysis steps is stored in an interchange format, which is parsed to instantiate an actual learning setup. This format includes the following information:

- A location of the target system
- An instance pool of predetermined data values (such as credentials)
- A description of the alphabet, i.e., a list of methods that are to be invoked
- For every method information the symbolic names of parameters and return values

Such of a setup description file concerned with learning the example WSDL e-commerce application is presented in Figure 6.

The location of the target system is provided in Line 2, which denotes a URL from which to retrieve the WSDL interface descriptor. From this descriptor, tools such as `wsimport` can fully automatically generate Java proxy classes, which can be employed by a configurable test driver to facilitate SUL invocations.

Lines 3 to 6 specify an instance pool of two string values which represent authentication credentials for the target system. By their very nature, such values have to be provided beforehand, i.e., have to be present in the instance pool.

The provided credentials are utilized in Lines 8 to 18, where a symbol for the `openSession` method of the SUL is defined. This method is parameterized, expecting the credentials previously defined for the instance pool. The execution result is stored in a variable as defined in Line 17.

The method `getAvailableProducts`, defined in lines 20 to 23, is simpler in comparison, as no parameters are expected. The most sophisticated symbol declaration is the one of `addProductsToShoppingCart`, where the second method

[3] `http://clarkparsia.com/pellet`

```
<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>
```

**Figure 6.** Example of a setup description file for automated setup instantiation

parameter can retrieve valuations from two different named variables: `productArray`, which is returned by the `getAvailableProducts` symbol, or `shoppingCart`, which is provided by a symbol not visible in the chosen excerpt of the setup descriptor. Each of those two variables indeed provides collections of values of the required type `Product`, whereas the method parameter only expects a single `Product` object. Thus the `elementOf` selector is applied onto the respective fields of the data structures, retrieving a singular data value.

## 8 Usage in LearnLib

The main class for interfacing the above description of learning setup in a LearnLib application is the class `LearnConfig`. Upon construction it receives the XML file name, and provides the deduced information, such as the learning alphabet, in a form compatible with the LearnLib API.

Figure 7shows how such an automatically generated test driver is used in a LearnLib scenario. In lines 1–2, the `LearnConfiguration` object is created from the path name of a learning setup descriptor. For interfacing the target system (which is assumed to be a web service), a dynamic proxy object of type `WSDLDynamicProxy` is instantiated (line 3). The purpose of this object is to provide a simple interface for invoking operations by name, which is achieved by generating proxy classes using the `wsimport` tool from the WSDL description of the service.

```
1 LearnConfiguration config
2   = new LearnConfiguration(new FileInputStream("learnsetup.xml"));
3 DynamicProxy proxy = new WSDLDynamicProxy(config.getServiceURL());
4 MembershipOracle mqOracle
5   = new ProxyOracle(config.getContextSeed(), proxy, ERROR, ERROR);
6
7 LearningAlgorithm learner = new Angluin();
8 learner.setAlphabet(config.getAlphabet());
9 learner.setOracle(mqOracle);
10
11 for(;;) {
12   learner.learn();
13   Automaton hypothesis = learner.getResult();
14   // ...
15 }
```

**Figure 7.** Using an automatically generated test driver for a webservice in LearnLib

As has been noted in Section 5, a component answering queries has to implement the `MembershipOracle` interface. In our scenario, this is the `ProxyOracle`

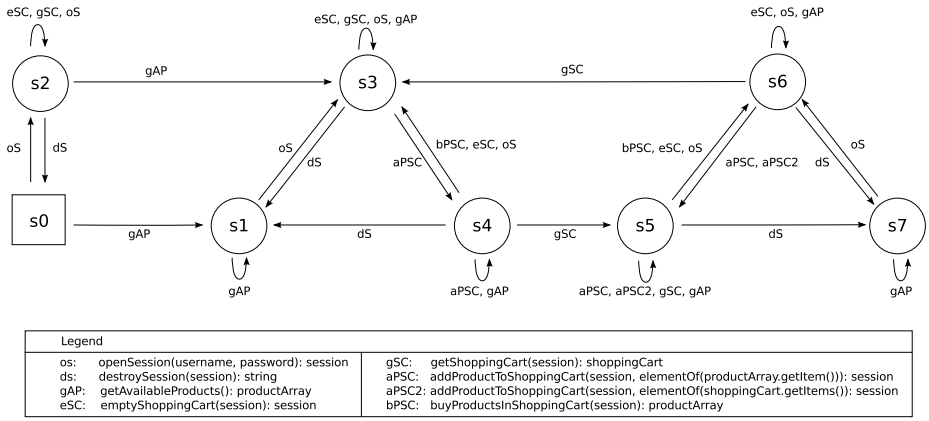


**Figure 8.** Learned model of an e-commerce application, learned with the setup descriptor of Figure 6

(lines 4–5). This oracle translates symbols of a special form to invocations on the `proxy` object. The `LearnConfiguration` method `getAlphabet()` (line 8) provides a learning alphabet which consists of symbols of the required form. The single symbols are created from the setup description in the fashion sketched in Section 7. Having instantiated a membership oracle along with a compatible learning alphabet, learning can be performed in the usual fashion with an arbitrary learning algorithm such as Angluin's $L^*$.

Figure 8 shows the result of executing a learning setup with the presented configuration. The result of executing a learning setup with the presented configuration is shown in Figure 8. This model reveals properties of the system's behavior, for instance how to finally place an order (which requires a non-empty shopping cart), which is useful behavioral information when trying to interact with the system.

The overall workflow for active automata learning within the presented framework, formalized in XPDD [9], is shown in Figure 9: on the left hand side input artifacts are visualized (e.g., the SUL's interface description), while on the left hand side output artifacts are visible (most importantly the learned model). If all processing steps (shown in the center of the figure) are automated, complete learning setups can be instantiated and executed without manual intervention.

## 9 Conclusion

Test drivers with mapper functionality are essential components of pretty much every active learning setup involving real-life systems that react according to data generated at runtime.

In this paper we presented how to manually create application-specific data-aware test drivers for LearnLib, an extensible framework for automata learning.

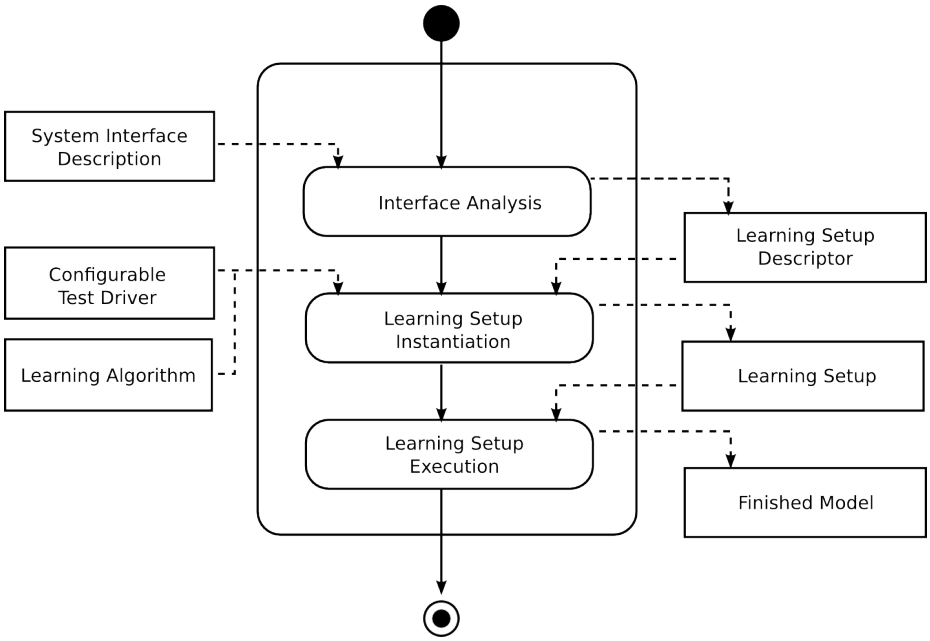


**Figure 9.** Overall workflow for (semi-)automated active automata learning

This is a straightforward process for systems of limited size, supported by the component-based approach of the LearnLib library.

For large-scale application and to create flexible learning setups, however, the approach of hand-crafting test-drivers is of limited appeal. Thus we presented a general architecture and concrete implementation of a reconfigurable test driver. The setup configuration for this test driver is generated by means of interface analysis, either conducted manually or (preferably) by automated means.

By introducing means to automatically generate setup descriptions, it is expected that automata learning becomes a much less laborious process, making adoption for real-life scenarios routinely feasible.

## References

1. Fides Aarts, Julien Schmaltz, and Frits W. Vaandrager. Inference and Abstraction of the Biometric Passport. In Margaria and Steffen [10], pages 673–686.
2. Antonia Bertolino, Paola Inverardi, Patrizio Pelliccione, and Massimo Tivoli. Automatic synthesis of behavior protocols for composable web-services. In Hans van Vliet and Valérie Issarny, editors, *ESEC/SIGSOFT FSE*, pages 141–150. ACM, 2009.
3. Therese Bohlin, Bengt Jonsson, and Siavash Soleimanifard. Inferring compact models of communication protocol entities. In Margaria and Steffen [10], pages 658–672.

4. Georges Bossert, Guillaume Hiet, and Thibaut Henin. Modelling to Simulate Botnet Command and Control Protocols for the Evaluation of Network Intrusion Detection Systems. In *Proceedings of the 2011 Conference on Network and Information Systems Security*, pages 1–8, La Rochelle, France, June 2011.
5. Andreas Hagerer, Hardi Hungar, Tiziana Margaria, Oliver Niese, Bernhard Steffen, and Hans-Dieter Ide. Demonstration of an operational procedure for the model-based testing of cti systems. In *FASE*, pages 336–340, 2002.
6. Falk Howar, Bernhard Steffen, Bengt Jonsson, and Sofia Cassel. Inferring Canonical Register Automata. In Viktor Kuncak and Andrey Rybalchenko, editors, *VMCAI*, volume 7148 of *Lecture Notes in Computer Science*, pages 251–266. Springer, 2012.
7. H. Hungar, T. Margaria, and B. Steffen. Test-based model generation for legacy systems. In *Test Conference, 2003. Proceedings. ITC 2003. International*, volume 1, pages 971–980, 30-Oct. 2, 2003.
8. Bengt Jonsson. Learning of Automata Models Extended with Data. In Marco Bernardo and Valérie Issarny, editors, *SFM*, volume 6659 of *Lecture Notes in Computer Science*, pages 327–349. Springer, 2011.
9. Georg Jung, Tiziana Margaria, Christian Wagner, and Marco Bakera. Formalizing a Methodology for Design- and Runtime Self-Healing. *Engineering of Autonomic and Autonomous Systems, IEEE International Workshop on*, 0:106–115, 2010.
10. Tiziana Margaria and Bernhard Steffen, editors. *Leveraging Applications of Formal Methods, Verification, and Validation - 4th International Symposium on Leveraging Applications, ISoLA 2010, Heraklion, Crete, Greece, October 18-21, 2010, Proceedings, Part I*, volume 6415 of *Lecture Notes in Computer Science*. Springer, 2010.
11. Maik Merten, Falk Howar, Bernhard Steffen, Sofia Cassel, and Bengt Jonsson. Demonstrating learning of register automata. In Cormac Flanagan and Barbara König, editors, *TACAS*, volume 7214 of *Lecture Notes in Computer Science*, pages 466–471. Springer, 2012.
12. Maik Merten, Falk Howar, Bernhard Steffen, Patrizio Pellicione, and Massimo Tivoli. Automated Inference of Models for Black Box Systems based on Interface Descriptions. In *ISoLA 2012*. to appear.
13. Maik Merten, Bernhard Steffen, Falk Howar, and Tiziana Margaria. Next Generation LearnLib. In *Seventeenth International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2011)*, 2011.
14. Harald Raffelt, Tiziana Margaria, Bernhard Steffen, and Maik Merten. Hybrid test of web applications with webtest. In *TAV-WEB '08: Proceedings of the 2008 workshop on Testing, analysis, and verification of web services and applications*, pages 1–7, New York, NY, USA, 2008. ACM.
15. Harald Raffelt, Maik Merten, Bernhard Steffen, and Tiziana Margaria. Dynamic testing via automata learning. *Int. J. Softw. Tools Technol. Transf.*, 11(4):307–324, 2009.
16. Muzammil Shahbaz, K. C. Shashidhar, and Robert Eschbach. Iterative refinement of specification for component based embedded systems. In Matthew B. Dwyer and Frank Tip, editors, *ISSTA*, pages 276–286. ACM, 2011.
17. W3C. Semantic Annotations for WSDL and XML Schema. Technical report, 2007. `http://www.w3.org/TR/sawsdl/`.