technische universität
dortmund

Bachelor Thesis

# Simplicity-Oriented Web-Based Control of Active Automata Learning

Alexander Bainczyk
April 2015

Supervisors:
Prof. Dr. Bernhard Steffen
Dr. Johannes Neubauer

TU Dortmund University
Department for Computer Sience
Chair for Programming Systems (Chair 5)
http://ls5-www.cs.tu-dortmund.de

# Contents

# 1 Introduction

The amount of available web applications on the Internet grows each day and so does the complexity and functional range of them. Companies such as Google and Microsoft offer their office suites as applications in the web browser which have a similar functional range as their desktop counterparts. With the use of modern web technologies tasks such as table calculations, text processing, the creation of presentations and many more are made available for every connected device that has access to the Internet through a web browser.

Many web applications have to store sensitive data about users, like e-mails, passwords and credit card information. As a consequence, developers are responsible for the safety of these data and have to make sure their web applications work exactly as they are intended. An error or a misbehavior could lead to a leak of sensitive user information. For example, when a user registers in an online payment service and then logs into it, he should only be able to manage his own account.

Beside the actual web application, many sites also offer interfaces for their service, on which other applications build and rely on. One example is the Twitter API. There are many clients solely for the Android platform listed in the Google Play Store. When the Twitter API does not work as intended, none of these applications fulfill their purpose either.

With modern and agile development methods such as SCRUM, release cycles of applications are shortened. In the permanent competition with other companies, more and more features are added with each version. Each change in the code holds the possibility of exposing bugs to the outside that can either make the application useless or in the worst case, expose user data. In order to prevent this, continuous quality control of software is necessary. Procedures such as static code analysis or unit tests help to find errors on the source code level but can not be used to verify correct behavior of an application in live operation. Further approaches like model-based testing allow to generate tests for a system automatically [30]. This, on the other hand, requires that a model that describes the system already exists, which causes problems in case no specifications are given. This is the point where active automata learning comes in, a procedure that is dealt with in this work to generate models from black-box systems.

Finally, developers have to ensure that applications work exactly as intended through continuous testing in order to find and remove misbehaviors through all stages of the development process in order to keep errors at a minimum.

## 1.1 Motivation and Background

In the context of a group project in the summer term of 2014 at the TU Dortmund University a web application with an accompanying web service has been developed by students of the lecture *Webtechnologien 2*. This application allows the management of todos and its correct behavior has been validated by writing unit and integration tests as well as by applying active automata learning. For this purpose, the framework LearnLib has been used by manually creating test cases which require an examination of the framework and programming knowledges in Java.

Oftentimes, when developing software products in teams, the area of responsibility of involved people differs. The actual development of an application and the quality control may not necessarily be executed by a single person (figure 1.1 (a)) but by people with different expertises. This leads to the problem that those who are responsible for the quality control of a product may not familiar with Java and therefore are not able to setup learning experiments with the LearnLib. Thus, the advantages of using active automata learning can not be applied in those cases.

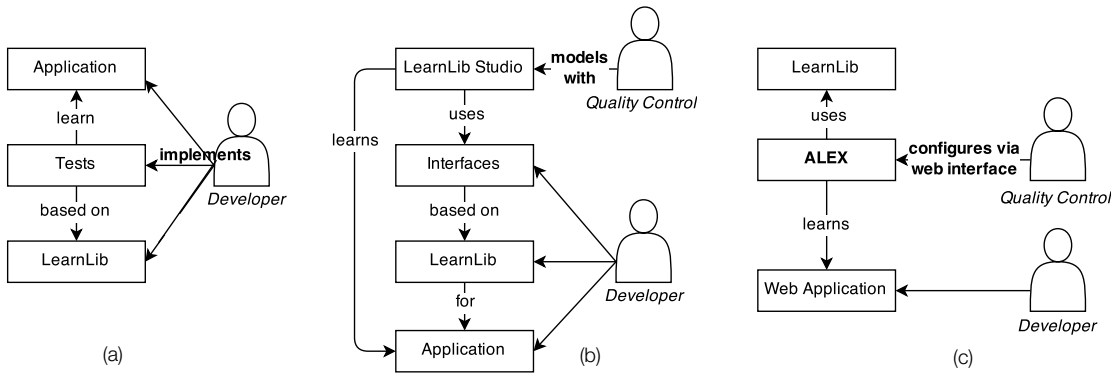


**Figure 1.1:** Creation of test cases: (a) Manual implementation. (b) Modeling with LearnLib Studio. (c) Configuration with ALEX

As a consequence, the demand of tools that allow learning applications without having a technical background is given. Existing software such as LearnLib Studio have a graphical user interface and offer a process-oriented approach to active automata learning, so that people without programming knowledges are able to apply this procedure. Still, Java code that interacts with the LearnLib has to be implemented in order to make an application's functions available for the use with LearnLib Studio. This approach is flexible, but adds a further intermediate step between the development and the testing process which costs

time and therefore may delay the release cycle of a software (figure 1.1 (b)). As a result, the idea is to remove this step and allow learning applications without the requirement of either knowing the LearnLib or Java. Due to the large amount of possible use cases it is concentrated on finding solutions for simplifying active automata learning of web applications and web services (figure 1.1 (c)).

This thesis takes on existing ideas and solutions [30] for the graphical modeling of learning experiments and alphabets that are used for active automata learning and refine them.

## 1.2 Goals of this Work

The focus of this work is the exploration and illustration of methods to simplify active automata learning for end users. The goal is to minimize the effort to familiarize with the LearnLib and to create possibilities that make the technical knowledge of writing software with Java unnecessary. Therefore, an application with a graphical user interface is developed which enables users to control core aspects of active automata learning of web applications and web services. In order to reach this goal, three consecutive steps are made:

1. **Exploration of methods for the simplification of active automata learning**
   For this matter, existing solutions are presented, their advantages and disadvantages discussed and potential approaches for their simplification given.

2. **Development of a web-based application that implements these concepts**
   After the possibilities are found, an application is developed in cooperation with Alexander Schieweck [25] who has build a RESTful web service for the LearnLib. In this thesis a single-page web application is introduced that communicates with this service in order to access functions of the framework. In this context, different frameworks and libraries are presented and evaluated from the standpoint of their usefulness with respect to the defined goals.

3. **Evaluation of the developed solution**
   Finally, the developed application is evaluated. Therefore, it is compared to existing solutions ant then used to model and execute a learning experiment on a web application. Then, conclusions are drawn relating to its convenience to use and its practical relevance for active automata learning.

## 1.3 Structure

In the second chapter, the preliminaries for understanding this document are stated. On the one hand, this includes the presentation of the characteristics of web applications and

web services, as well as the presentation of ideas of RESTful interfaces and single-page web applications. Finally, an introduction to the theory of active automata learning and its practical appliance is given.

Chapter 3 is about existing tools for this method that are dealt with in the scope of this thesis. It is followed by a description of the developed application ALEX in chapter 4. This refers to the representation of ideas and concepts in the first part and the details of implementation in the second part. Chapter 5 addresses the evaluation of the application. In the last chapter, the results are brought together, highlights are pointed out and a perspective on further development is given.

# 2 Preliminaries

The contents of this chapter communicate the base for understanding the ideas and concepts presented in this thesis. First, the characteristics of web applications, web services and the communication with those are presented. Then, an introduction to the theory of active automata learning and its practical appliance for the mentioned use cases follows. Finally, a note on single-page web applications, its concepts and used programming patterns is given.

## 2.1 Web Applications

Web applications can be described as a variant of the client-server-model where one or multiple clients request data or services from a server that provides them [8]. Client and server can thereby run on the same or on different computers. The communication between both parts is based on a use case specific but defined protocol where a client sends a *request* to the server that processes it and sends back a *response*.

As described in [30], in web applications, a web browser that is handled by a user acts as the client and runs on Internet-connected devices like computers and smartphones. A locally or remotely installed web server functions as the server part in this model. They communicate over protocols such as Hypertext Transfer Protocol (HTTP) and HTTP Secure (HTTPS), where a client sends an HTTP request to a server. The server processes the query and answers with an HTTP response which can for example contain a static Hypertext Markup Language (HTML) file that can then be displayed in a web browser.

A common way to structure web applications is the three-tier-architecture, where every tier, respectively layer has a different task. On the top, a web server (presentation) receives HTTP requests from a web client and sends HTTP responses. An application server (business logic) processes requests and handles the application's logic. On the lowest layer (persistence), a database or another data persistence system is located. Typically in this model, a tier only communicates with the one above and the one below, what, theoretically, makes every layer replaceable.

Furthermore, web applications fall into the category of reactive systems [30]. That means they are characterized firstly by determinism, which says that its behavior for inputs has to be consistent for multiple executions. To fulfill this, the system has to be fully controllable without influences of third parties. Secondly, it has to run permanently

and process and react to multiple inputs simultaneously, what is called concurrency. The last aspect is named reliability, according to which reactive systems have to react reliably to inputs and, if possible, within a certain time span.

## 2.2 Web Services

In contrast to web applications, web services do not target web browsers only, but are designed for application-to-application communication in distributed environments [10]. The goal is to offer a language and platform neutral interface for communication and data exchange to make functionalities of a system available for a multitude of other systems. For example, a web service that is written in Java and hosted on a Unix-based system would be able to communicate with a PHP or a JavaScript based service that uses Windows as a server environment. For the data transfer, HTTP and HTTPS could be used again.

This approach demands a standardized format for the exchange of information. In modern web services, the data-interchange formats XML and JSON are typically used.

### Extensible Markup Language (XML)

XML is used to describe semi-structured data and can serve as an interchange format. Documents consist of *elements* that have opening and closing tags *<element></element>* or are self-closing *<element />*. They can be nested and a document must have a root element that contains all others. [2]

### JavaScript Object Notation (JSON)

JSON is a lightweight, language-independent and text-based data-interchange format. Its syntax is based on JavaScript's object literals and therefore it supports the representation of arrays, objects, strings, numbers, boolean values and the null value. [11]

Figure 2.1 exemplarily shows a possible exchange of data between a client and a server with different languages.

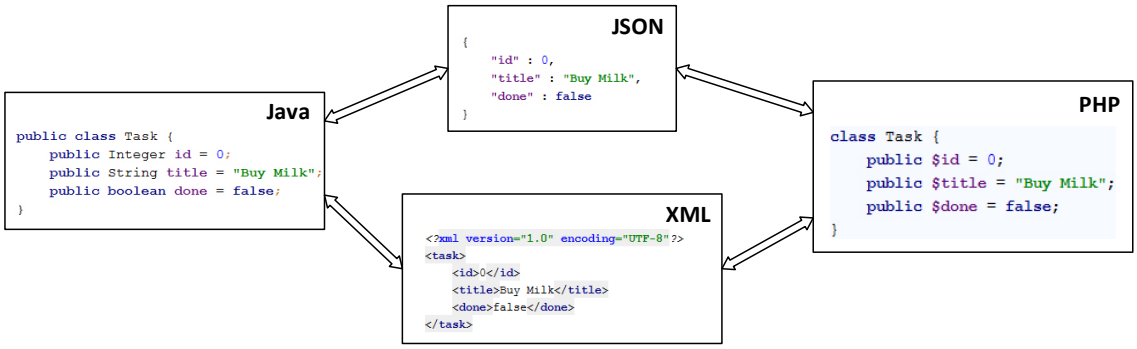


**Figure 2.1:** Exchange of a Task object between Java and PHP with XML and JSON

In order to answer the question which format should be used when developing client-server applications, Nurseitov, Paulson, Reynolds et al. compared both in terms of use of resources and transmission time [23]. The results state that generally, the use of JSON outperforms XML in both categories.

## 2.3 RESTful Web APIs

Representational State Transfer (REST) has first been introduced by Roy Fielding in his dissertation [13] as an architectural pattern for sharing information in distributed hypermedia systems. It is not a protocol for network-based communication like Simple Object Access Protocol (SOAP) [1] but rather a design pattern with best practices.

A key term is *"stateless communication"*, which refers to the way state is handled. A server should not save any session information about its clients. Instead, a client must hold and send all data that is required for the server to understand a request.

In REST, any kind of information is abstracted as a *resource* that is identified by a *resource identifier*. It can, among other things, be structured data like JSON and XML, unstructured data like images and text files or other services.

A RESTful web service therefore makes its resources for other systems accessible through an Application Programming Interface (API) that is designed with REST. A resource is accessible via an Uniform Resource Identifier (URI) that can for example be an absolute path to a remotely located text file like *"http://website.com/resources/files/file.txt"*. In many cases Create, Read, Update, Delete (CRUD) operations should be executed on a resource. For this purpose, best practices exist that can be extracted from [6]. An example for how this can be achieved with a to-do object as a resource is presented in table 2.1. A list of all available HTTP methods and its meaning can be found at [5].

| **HTTP method** | **URI** | **Description** |
|---|---|---|
| POST | .../rest/todos | Creates a new to-do |
| GET | .../rest/todos/{id} | Reads an existing to-do with id {id} |
| PUT | .../rest/todos/{id} | Updates an existing to-do with id {id} |
| DELETE | .../rest/todos/{id} | Deletes an existing to-do with id {id} |

**Table 2.1:** CRUD operations on a HTTP-accessible resource

As it is shown in figure 2.2, an HTTP request consists of a Request-Line, a General-Header, a Request-Header, an Entity-Header and a Message-Body. In a Request-Line the triple of HTTP method, URI and HTTP version is specified. The field *Accept* defines the accepted type of the resource. Depending on its value a server could decide on whether for example a JSON or an XML representation of it is sent back. The Message-Body can contain data that is sent to the server which can be used in POST and PUT requests to create or update a resource.

| | |
|---|---|
| Request-Line | GET /tasks/all HTTP/1.1 |
| General-Header | Host: localhost:8080<br>Connection: keep-alive |
| Request-Header | Accept: text/html,application/xhtml+xml<br>User-Agent: Mozilla/5.0 (Windows NT 6.3; Win64; x64) ...<br>Accept-Encoding: gzip, deflate, sdch |
| Entity-Header | *empty for GET requests* |
| *blank line* | |
| Message-Body | *empty for GET requests, may contain data for PUT/POST requests* |

**Figure 2.2:** The structure of an HTTP GET request

The difference between an HTTP response and a request is that the Request-Line is replaced by a Status-Line which in this example contains the string *HTTP/1.1 200 OK*. It defines the used protocol and a status code. These codes lay in the range from *1xx* to *5xx* [5] , and are used to communicate whether an operation succeeded or failed. When for example the creation of a resource with a *POST* request succeeds, the server might send the status code *201 (Created)*, so that the client knows about the result of a request and can react accordingly. Codes in the range *4xx* indicate client side errors, like the commonly known *404 (Not Found)* code, indicating a wrong URI has been specified. The Message-Body contains the string representation of the requested resource, here it would be an HTML document of a website.

## 2.4 Single-Page Web Applications

A Single-Page Application (SPA) is a program whose application logic is written in JavaScript and that uses HTML for the presentation. In contrast to classic websites where content is normally spread across multiple pages, an SPA works on a single HTML file. All required contents and resources are loaded on the first request or on demand without the need to reload the page. A goal of an SPA is to have a fully functional application running in the browser that behaves like a native one. Another point is that they can be designed to run as a client for a web service that allows the management of available resources via a graphical interface. [22]

An advantage of using SPAs is the platform independence of these applications. They can theoretically run on every device that has a web browser which allows the execution of JavaScript. This eliminates the effort of a local installation and prevents incompatibilities with the underlying system. With frameworks such as *PhoneGap*[1] and *Node-Webkit*[2] JavaScript applications can even be created for a range of mobile platforms like iOS and

[1] `http://phonegap.com/`
[2] `http://nwjs.io/`

Android and at the same time run on Windows, Linux and MacOS while maintaining a single or minimally varying code base.

To synchronize data with a web service or to load required resources, JavaScript offers the creation of Asynchronous JavaScript and XML (AJAX) requests. These are normal HTTP requests that are handled asynchronously (figure 2.3) so that the application continues to work without blocking user interactions. The *XML* part from the name refers to earlier times, when XML dominated the client-server communication as an exchange format.

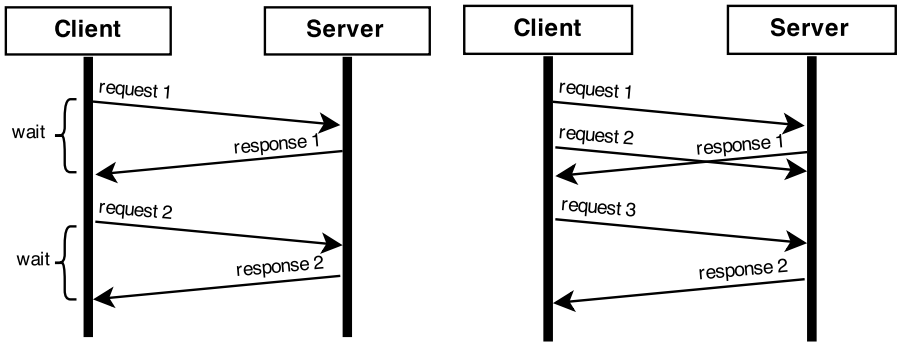


**Figure 2.3:** Synchronous (left) and asynchronous (right) request handling

The default AJAX specification determines that requests must be made within the same domain. So, a request that is sent from *http://localhost/app/*, may only access servers from *localhost* due to the same domain policy. That is to prevent Cross-Site-Request-Forgery (CSRF) [26] attacks but causes problems when relying on data from other APIs. Cross-Origin Resource Sharing (CORS) [29] bypasses this policy by defining allowed origins and request methods in HTTP header fields that are sent with each HTTP response from a server. The web browser then knows that it is allowed to communicate with a specific server and therefore passes outgoing requests and incoming responses. That is especially useful in cases where an SPA relies on resources of third party web services.

When writing SPAs the persistence of data is of another concern. There are use cases of SPAs that do not rely on any kind of storage, like *TiddlyWiki*[3]. This is a JavaScript-based Wiki software that writes changes directly into the HTML document and saves the new state by downloading itself. Oftentimes, it is desired to have a persistence layer. Technologies such as *SessionStorage* and *LocalStorage* that have been introduced as a part of the HTML5 specification [4] take over that role. Both solutions concentrate on storing client-side data in key value pairs where the value is a JSON object. The difference between these is the life span of the stored data. The SessionStorage keeps data alive in the same tab as long as it or the browser is opened. Otherwise, objects that are stored

[3] `http://tiddlywiki.com/`

in the LocalStorage can be accessed in all tabs of a browser and are retrievable even after a restart. So, the data has to be deleted manually. A third solution is to make use of a server-side storage systems like databases, where data is sent to and fetched from a server via AJAX requests. Client-side and server-side techniques can also be combined to use applications offline that synchronize data with a server as soon as a network connection is available.

## 2.5 Active Automata Learning

If specifications of a system, where little to no knowledge about its inner structure is known (black-box), but its reactions to inputs are observable, automata learning can be used to generate a model from the outputs of a system via testing. A generated model can be presented as automaton and ideally shows the same behavior as the System Under Learning (SUL). It can, for example, be used to validate correct behavior or observe changes of a system. Due to the characteristics of black-box systems it can never be made sure that the created models represent a system. [30]

Automata learning can be distinguished into a *passive* and an *active* variant [21]. Given the case there already exists data of a system that describes its input and output behavior, algorithms for passive automata learning may analyze the training set and build a model from it. A disadvantage of this method is, if the data is incomplete, the model does not represent the complete SUL either.

In contrast to that, algorithms for active automata learning generate data by themselves by performing actions to a system and observing its reactions. They are analyzed and used to create a model. Generally, in order to apply active automata learning it is required that an SUL shows (input) deterministic behavior [28]. This premise is required, so that a learning algorithm can assume to have generated a model that represents the SUL and can stop conducting inputs to it. In practice, reactive systems can be modeled as Mealy machines.

**Definition 1 (Mealy Machine)**
*A Mealy machine is defined as a tuple* $M = (Q,\Sigma,\Omega,\delta,\lambda,q_0)$ *where*

- $Q$ *is a finite nonempty set of states*
- $\Sigma$ *is a finite input alphabet*
- $\Omega$ *is a finite output alphabet*
- $\delta : Q \times \Sigma \to Q$ *is the transition function*
- $\lambda : Q \times \Sigma \to \Omega$ *is the output function*
- $q_0$ *is the initial state*

According to [28], active automata learning can be separated in two alternating phases. The first one is called *exploration phase*, during which Membership Queries (MQ) are

conducted to an SUL. For an MQ a word $w \in \Sigma^*$ is executed on the SUL which triggers an output that is collected and analyzed. In practice, words consist of sequences of real interactions with a system such as clicking on elements of a user interface. A learning algorithm then constructs a hypothesis model based on the observations. In the second phase, also called *testing phase*, the learned model is compared to the actual system with the goal to check whether the hypothesis represents the SUL. Therefore, Equivalence Queries (EQ) are used to find words where the output of the SUL differs from the one of the hypothesis. A differentiating word is called *counterexample* and is then passed to the learning algorithm that refines the learned model under respect of the counterexample in the next iteration. Usually, the iteration stops when no more counterexamples are found.

Furthermore, it is required that MQs are independent from each other. The outcome of this is that systems have to be transferred into a state where past MQs do not influence the current one. This is achieved with so called *reset* mechanisms that can vary from use case to use case. In the context of web applications for example a reset could be done by clearing a database. [28]

### 2.5.1 The Algorithm $L^*_M$

Typically, when given an introduction to active automata learning, Dana Angluin's algorithm $L^*$ [7] is mentioned. It has originally been developed for Deterministic Finite Automata (DFA) but for modeling reactive systems, the extension $L^*_M$ is described in [19] for inferring Mealy machines.

The algorithm internally uses a so called *observation table* from which the hypothesis model is build and which holds the results for conducted MQs to an SUL. It presents the sets for prefixes $S$, $S\Sigma$ and a set of suffixes $E$. For every word $se$ with $s \in S \cup S\Sigma$, $e \in E$ the table saves the outcome of an MQ. An example for a table and the corresponding hypothesis can be seen in figure 2.4.

| | | $E$ | |
|---|---|---|---|
| | | login | logout |
| $S$ | $\epsilon$ | OK | FAILED |
| | login | FAILED | OK |
| $S\Sigma$ | logout | OK | FAILED |
| | login login | FAILED | OK |
| | login logout | OK | FAILED |

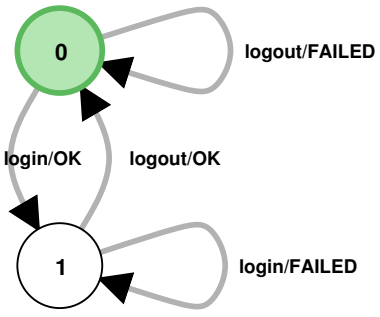


**Figure 2.4:** Observation table (left) with output symbols *OK*, *FAILED* and hypothesis (right)

In order to create a Mealy machine from an observation table, it has to be *closed* and *consistent* [27]. The rows that belong to $S$ represent the sequences to established states

whereas $S\Sigma$ belongs to the successors states. Let $[s]$ indicate the state for the sequence $s$, then the initial state of the hypothesis is $[\epsilon]$. Transitions are made from $[s]$ to $[se]$ for all $s \in S$ and $e \in E$ with the label of the output from $se$.

During the ongoing research for more space and time efficient algorithms, a few have emerged as alternatives to the $L^*$ algorithm. Representatives of these are the Direct Hypothesis Construction (DHC) [20], the Observation Pack [15] and the TTT [14] algorithm.

### 2.5.2 Discrimination Trees

The data structure of a discrimination tree has been introduced by Kearns&Vazirani [16] in the context of active automata learning and their developed algorithm. It is also the main data structure of the Observation Pack and the TTT algorithm for saving observations of conducted MQs in an efficient, redundancy-free manner as an alternative to observation tables.

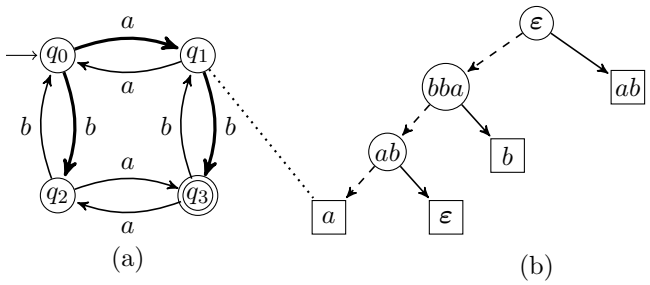


**Figure 2.5:** (a) Possible hypothesis DFA (b) Possible discrimination tree Source: [15]

A discrimination tree as shown in figure 2.5 is a rooted binary tree where leaves are labeled with *access strings* $s \in S \subset \Sigma^*$ and inner nodes with *distinguishing strings* $d \in \Sigma^*$, starting from the root with the empty word $\epsilon$. Each leaf represents a unique state of an SUL that is reached by executing $s$ from its start state. For each node with a label $d$ applies that $\exists\ (s, s') \in S$ with $s \neq s'$ so that an SUL accepts $sd$ and rejects $s'd$ or vice versa. The accepted string then represents the left child whereas the rejected one represents the right child of the node.

A hypothesis model is obtained as follows: first, states are created that are labeled with access strings from a tree. Then, for each $(s, a) \in S \times \Sigma$ a so called *sift* operation is executed with $sa$. Therefore, an MQ with the word $sad$ is conducted where $d$ is the distinguishing string of a node of a tree, starting with the root. If the MQ is accepted, the procedure is continued with the right child, otherwise with the left one, until a leaf with an access string $s'$ is reached. Finally, a transition from the state with label $s$ to the one with label $s'$ is made. The edge is labeled with $a$.

A deeper insight in the algorithm, an explanation how counterexample are handled and runtime and correctness proves can be found at [16] and [15].

### 2.5.3 Learning Web Applications and Web Services

When developing an application, normally tests are written in order to validate its functionalities. Most common methods for testing are unit tests and integration tests. While unit tests are designed to test the input and output behavior of single functions, integration tests test the interaction of functions through combination of modules. With the approach of learn-based testing, the creation of single test cases is not required since a learning algorithm takes over this step automatically. Also, in cases where no formal specification of a system is given, changes can be observed by comparing versions of learned models over time. [30]

# 3 Tools for Active Automata Learning

Different tools are developed for the practical appliance of active automata learning methods. They make the previously described theory available for developers and testers of real world applications. In the following, two projects are presented that are worked with in the context of this document.

## 3.1 LearnLib

LearnLib[1] [24] [28] is a Java framework for active automata learning that is developed at the Chair of Programming Systems at the TU Dortmund University. It offers a wide range of related algorithms and data structures. It is designed to allow learning various applications by giving developers interfaces for custom alphabet construction, utilities for connecting SULs with learn algorithms, as well as different kinds of so called oracles.

**Membership Oracle**

A membership oracle receives a word, executes its symbols on an SUL and returns observed reactions for example in form of another word.

**Equivalence Oracle**

An equivalence oracle is another interface of the LearnLib that receives a hypothesis and an alphabet and conducts EQs to an SUL. If it is assumed that the SUL is equivalent to the model it returns null, otherwise a counterexample. Queries are mostly approximated by using MQs and the LearnLib offers different strategies for that.

## 3.2 LearnLib Studio

As described in [21], LearnLib Studio is developed as a graphical user interface (figure 3.1) for the LearnLib. It is based on the graphical process modeling framework Java Application Building Center (jABC)[2] and allows the design, execution and analysis of simple to complex learning experiments as a graph model. These setups typically consist of a configuration, a learning and a result analysis part.

[1] `http://learnlib.de/`
[2] `http://hope.scce.info/`

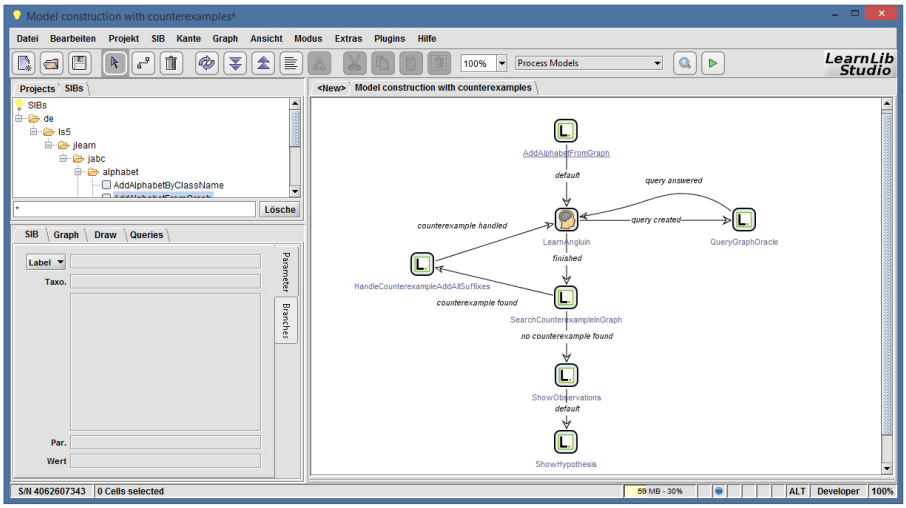


**Figure 3.1:** The user interface of LearnLib Studio

A graph consists of multiple Service-Independent Building Blocks (SIB) that model independent tasks in the context of automata learning. SIBs are connected with directed edges that mark the data flow from one SIB to another. LearnLib Studio offers a predefined set of SIBs that can be integrated in experiments. It can be chosen from different learning algorithms, equivalence oracles, SIBs for modeling an alphabet and more.

Furthermore, the software offers existing services which allow to easily collect statistics about learning processes like the amount of MQs, EQs and symbol calls. These numbers can then be displayed in charts.

Thanks to available interfaces, creating own SIBs with a custom behavior is also realizable. With this possibility, own symbols that contain input actions for a specific SUL can be created. This has the advantage that testers do not necessarily have to be able to write Java classes in order to test an application if the required SIBs have already been created.

In an attempt to bring the functionalities of the jABC and the LearnLib Studio to cloud based environments, WebABC is developed that is also described in [21]. It is a web based application with the same concept as the one shown above.

# 4 ALEX – A Simplicity-Oriented Approach

Current solutions for learning applications require users to acquire knowledge in handling the presented tools, an ability to write Java code and a certain setup time for creating input symbols. This leads to the idea of a further abstraction and simplification of certain tasks.

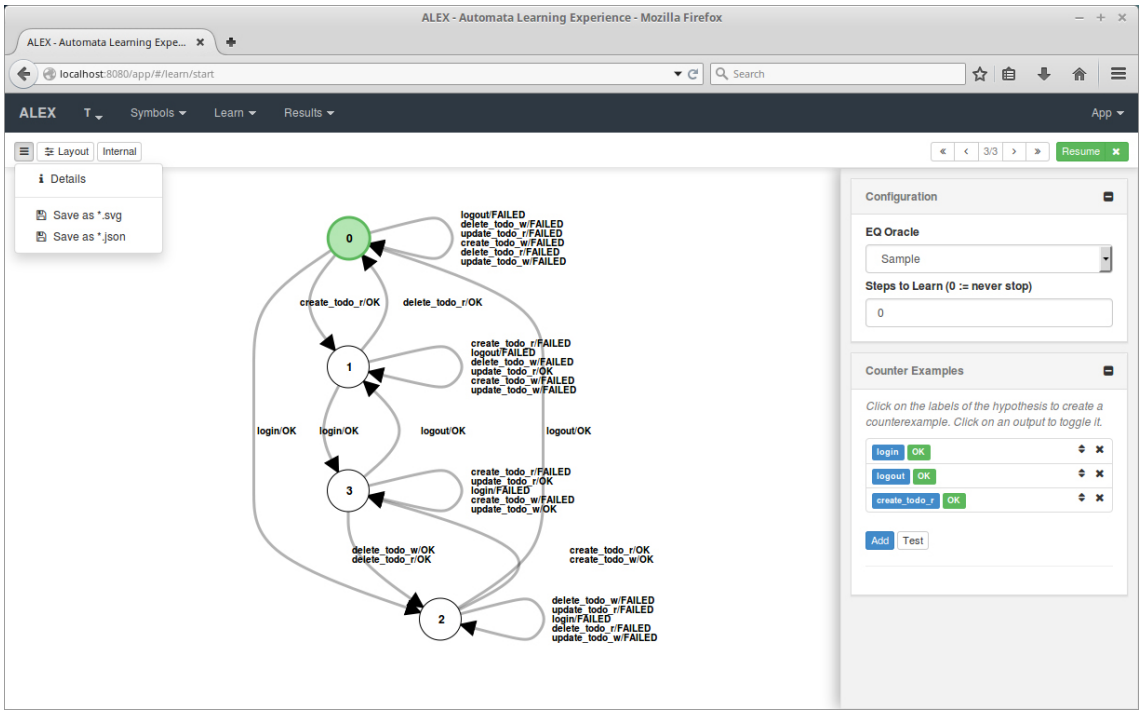


**Figure 4.1:** The front-end of ALEX

Automata Learning Experience (ALEX) is a two component solution which consists of a web service for the LearnLib and a web-based SPA as a graphical user interface on top of it. The web service is developed by Alexander Schieweck and is described in his bachelor thesis [25]. The front-end that can be seen in figure 4.1 is developed by the author of this work. The intention is to offer a simple to use interface for modeling alphabets, creating and evaluating learning experiments without the need to know Java, the LearnLib and third party libraries for learning web applications and web services. Instead of a manual approach, the main concept is form-based. This applies to both the symbol construction and the modeling of learning experiments.

This intention does have certain drawbacks. The generalization of the alphabet modeling in a browser interface results in a less flexible solution that does not cover all possible use cases for all applications. For every use case, an adapter that connects an SUL with the learner has to be written in Java. As a consequence, ALEX is build to support active automata learning of web applications and web services using Mealy machines. Another point is that the modeling of learning experiments is not as flexible as known from the LearnLib Studio, but reduced to core features. This being said, the developed software does not aim to replace existing, highly configurable tools, but to achieve results with less effort in comparable situations.

For creating such an application, ways to abstract these processes for web applications and web services as well as methods to create a simple to use interface has had to be found. The following sections dive deeper into the concepts of ALEX, while concentrating on the client-side part.

## 4.1 Used Technologies

For the development of the client of ALEX several technologies, frameworks and JavaScript libraries are used. In the following, only those that are deeply integrated are listed and described. Other ones that are used for the implementation of a specific feature are mentioned in chapter 4.3.

The developed application for controlling active automata learning makes use of the web technologies HTML, Cascading Style Sheets (CSS) for the markup and JavaScript for the logic and is intended to be used in a web browser. Because it functions as a graphical interface for the RESTful API of the LearnLib the use of a web-based solution has the advantage that users do not have to install further applications. The usage of JavaScript eliminates third party dependencies since required libraries are loaded directly with the application itself. As a result, it is not limited to a specific operating system. Thanks to front-end frameworks like Bootstrap, an optical appealing interface can be developed, that is hard to achieve with Java in combination with Swing, AWT or JavaFX. Another aspect is that the application itself does not do any CPU intense operations. So, it is distributed by the RESTful API as an independent resource. finally, ALEX is shipped as an executable jar file with an embedded version of the HTTP server and servlet engine jetty from the Eclipse Foundation[1]. On execution, the server for the web service starts and the front-end is accessible over a web browser.

[1] `http://eclipse.org/jetty/`

### 4.1.1 RESTful API for LearnLib

The server-side part of ALEX is a web service for LearnLib, written in Java and developed at the same time as the graphical client. Besides a persistence layer for saving symbols, hypotheses, statistics and further related data, it makes the functionalities of LearnLib available for third party applications by offering a JSON-based HTTP interface. It operates directly on LearnLib and thus allows learning web applications and web services with a given set of symbols. The learning process can be customized insofar as different algorithms and EQ oracles can be chosen.

### 4.1.2 AngularJS

AngularJS[2] is an open source JavaScript framework that is developed by Google. The big community and the growing number of third party libraries make it a good fit for developing SPAs. It offers functionalities that are mostly known from desktop applications, such as two-way-binding [17, p. 12–13], dependency injection [17, p. 151–152] and modularization of code. Furthermore, it is presented as an out-of-the-box framework to develop everything starting from a dynamic form and its validation up to complete applications.

A feature that makes it stand out from other client-side frameworks is the ability to create *directives*. They allow the creation of custom HTML elements or extending the functionality and the behavior of existing ones. As a result of this, independent components of a web page can be written once and reused in other parts of it or in other applications.

```
1  <script type="text/javascript">
2    angular.module('ALEX', [])
3      .directive('symbolList', function($http){
4        return {
5          scope: {
6            projectId: '@'
7          },
8          link: function(scope, element, attributes){
9            $http.get('/rest/projects/' + scope.projectId + '/symbols')
10             .success(function(response){
11               scope.symbols = response.data;
12             })
13         },
14         template: '<div ng-repeat="symbol in symbols">{{symbol.name}}</div>'
15       }
16     })
17 </script>
18
19 <symbol-list project-id="1"></symbol-list>
```

**Figure 4.2:** Creation of a custom HTML element in AngularJS

[2] https://angularjs.org/

The example in figure 4.2 illustrates how a custom HTML element is created that lists the names of all symbols of a project. In line 2 a module is created and the directive *symbolList* assigned to it in line 3. The directive requires the *$http* service that works as a wrapper for native AJAX requests. In AngularJS *scope* objects contain properties and functions that are available in a template. Lines 5–6 define which attribute values of the custom element are saved in the scope. The function *link* starting in line 8 describes the behavior of the directive. Here, an AJAX request is made to the API in order to get all symbols that are stored in a database. If the request succeeds, the symbols are assigned to the scope object. The contents of the *template* property show the HTML that is placed inside the custom element. In this example AngularJS creates a *div* element for each symbol object in *scope.symbols* and fills it with the name of the symbol. Finally, in line 19, the element that corresponds to the directive is instantiated. Figure 4.3 presents the use of it, the compiled template and the result in a web browser.

```
<h4>Symbols</h4>
<symbol-list
  project-id="1">
</symbol-list>
```

**(a)** Source

```
<h4>Symbols</h4>
<symbol-list
  project-id="1" class="ng-isolate-scope">
  <div ng-repeat="symbol in symbols"
    class="ng-binding ng-scope">Login</div>
  <div ng-repeat="symbol in symbols"
    class="ng-binding ng-scope">Logout</div>
  <div ng-repeat="symbol in symbols"
    class="ng-binding ng-scope">Register</div>
</symbol-list>
```

**(b)** Compiled


**(c)** Browser

**Figure 4.3:** AngularJS *symbolList* directive in action

The possibility to create dynamic multi page applications is realized with AngularJS routing ability. A Uniform Resource Locator (URL) can be mapped to a controller and an HTML template that is loaded once via AJAX, parsed and then rendered. Thanks to a build-in template cache another request to the same site would prevent the template loading one more time.

As a conclusion, AngularJS is used for the front-end of ALEX because the requirements demanded a dynamic solution for an interactive interface that is able to process a lot of user input, that reacts to users actions and plays well with RESTful APIs. Further functionalities of this framework can be gathered from [17].

### 4.1.3 Bootstrap

For the front-end of ALEX, the Sass[3] version of Bootstrap[4] is used. It is a component-based CSS framework which is drafted for developing cross browser and responsive[5] user interfaces. Beside a grid system, it offers predefined stylesheets for forms, buttons, navi-

[3] CSS extension language that is commonly used for preprocessing. `http://sass-lang.com/`

[4] `http://getbootstrap.com/`

[5] The presentation of a website responds to changes of the environment, e.g. display resolution and pixel density [12]

gations and many more which makes it a utility to quickly realize visually appealing user interfaces. Due to its availability for CSS preprocessors like Less and Sass, Bootstrap can be customized in many ways, allowing to remove unneeded stylesheets for smaller CSS files. The JavaScript library bootstrap.js lets developers extends the framework by offering behaviors for example for modal windows and dropdown menus.

### 4.1.4 Angular-UI

Angular-UI[6] offers multiple modules that extend and complement the AngularJS framework. In this work UI-Bootstrap[7] is used. It replaces the bootstrap.js library by implementing their own solutions so that its directives and services can be used. Furthermore, UI-Router[8] is used for URL routing purposes.

### 4.1.5 D3.js

D3.js[9] is a JavaScript library for data visualization and manipulation using Scalable Vector Graphics (SVG). This format is based on XML and is used to draw two-dimensional graphics. It can be embedded in HTML documents using the *<svg>* tag [3]. In ALEX, D3 is used to render hypotheses and discrimination trees in order to make them interactive.

### 4.1.6 N3-Charts

N3-Charts[10] is described as an AngularJS specific library for drawing SVG supported charts using D3. In the developed software, it is utilized to present statistics of learning processes.

### 4.1.7 Grunt

Grunt[11] is a task runner that automates certain parts of the development of JavaScript applications. It offers plug-ins for concatenating and compressing script files, stlyesheets and HTML templates. SPAs profit from the use of such tools because generally, all resources are loaded on the first request. So, minimizing the amount of those leads to a reduced use of network bandwidth which can be a critical point when relying on mobile data plans. Due to these measurements, ALEX fetches five files with about 1.3MB in size instead of about 3.6MB in 166 requests on startup. With browser-based caching being enabled, this number is reduced to about 100KB on further requests.

[6] `http://angular-ui.github.io/`
[7] `https://angular-ui.github.io/bootstrap/`
[8] `http://angular-ui.github.io/ui-router/`
[9] `http://d3js.org/`
[10] `http://n3-charts.github.io/line-chart/`
[11] `http://gruntjs.com/`

## 4.2 Concepts

Since ALEX consists of two parts, the tasks they handle also differ. As figure 4.4 shows, the back-end learns applications and is therefore linked to the LearnLib. Learning related objects like created symbols and generated models are persisted in a database. Through an API these functionalities are made available for clients, like the developed front-end. There, input symbols can be created, learning experiments modeled and evaluated and hypotheses displayed and interacted with graphically in a browser-based interface.

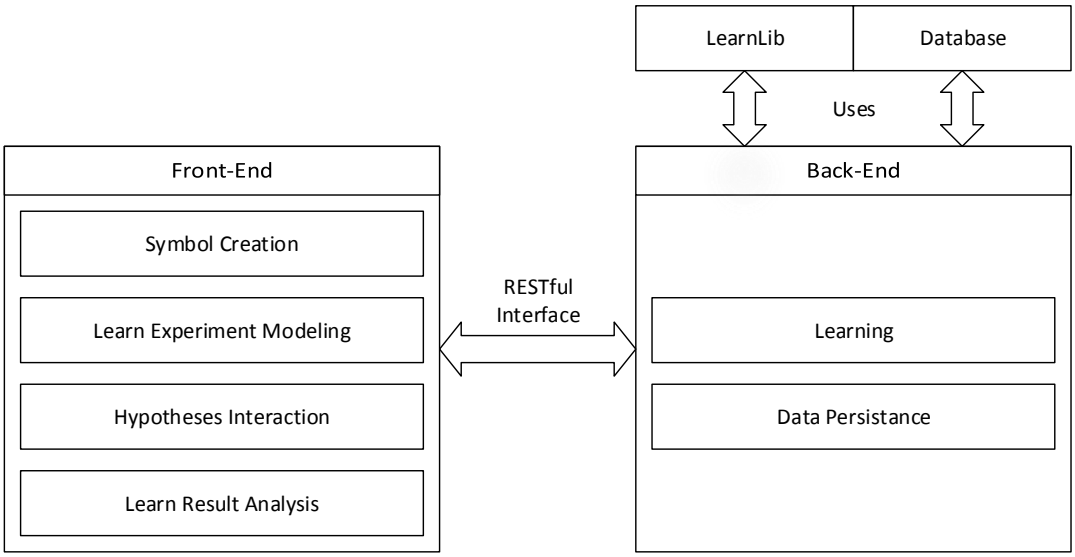


**Figure 4.4:** Interaction between and responsibilities of front-end and back-end in ALEX

The actual workflow of a user can be divided into three logical phases. In a preparation phase symbols for web applications and web services are created. Then, learning experiments can be modeled by defining a learning alphabet from the symbols and by specifying an algorithm in combination with an equivalence oracle. During the actual learning phase, the server learns the SUL and intermediate hypotheses can be displayed and interacted with in the browser. As a final phase, an evaluation of the results of previous learning processes is performed. This includes the displaying of statistics and the presentation and visual comparison of hypotheses and their internal data structures.

In the next sections the steps a user passes when making use of the front-end are described in detail on a conceptual basis.

### 4.2.1 Symbol Construction

The modeling of input symbols is an essential point for applying automata learning to an application. Manually creating symbols in Java that fit to an exact use case provides

flexibility but also proves to be a time consuming process and may require familiarization in third party libraries.

One approach to the simplification of this process is to reduce the flexibility that the manual approach comes with. This can be achieved by defining a limited set of executable actions that can be put into an SUL. The functionality of a symbol can then be build with different combinations of supported actions. The goal is to support enough of those which cope a high spectrum of interactions with web applications and web services. On the downside, the risk of not offering actions for all possible use cases exists. By now, this can only be dealt with by handing in more actions later. An effort has been made during the development process to offer a well selected set of those that are presented later in this section.

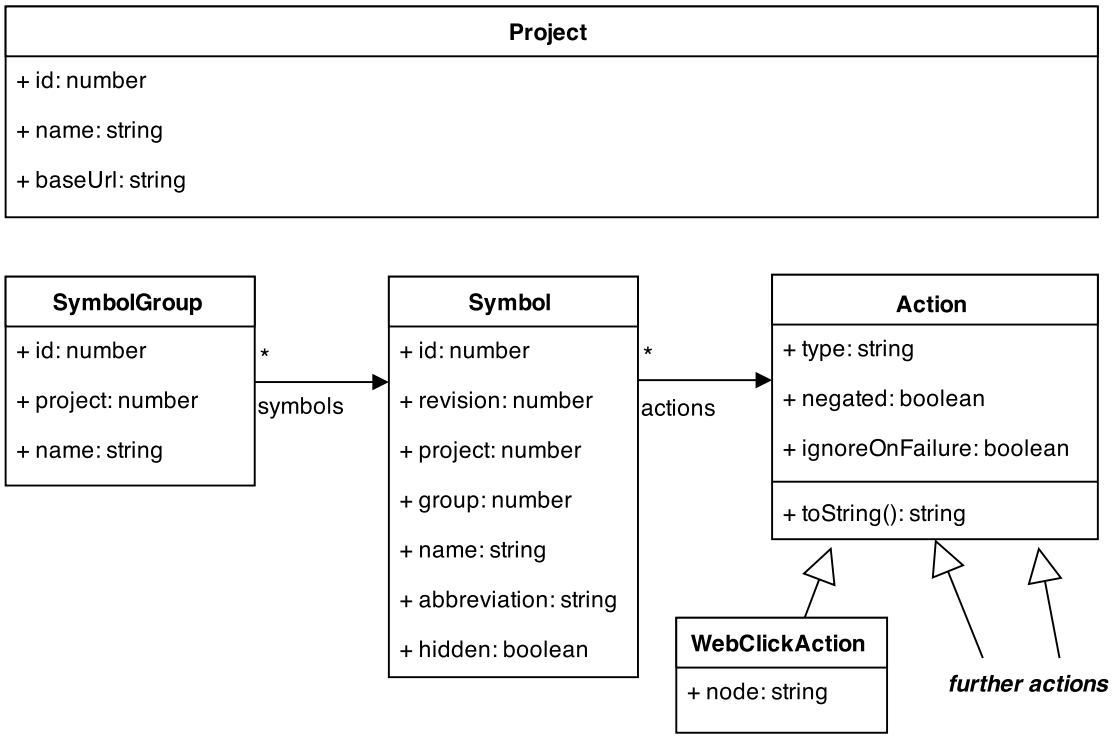


**Figure 4.5:** Class diagram for symbol encapsulation

Figure 4.5 presents the main objects that are involved in the symbol creation process as they are given by the front-end. A project serves as a container that holds all symbol groups and symbols that are used to learn an application. The property *baseUrl* therefore contains the URL an application to be learned is accessible under and should either start with *"http://"* or *"https://"*. There are no restrictions concerning the domain so that remotely installed applications can be learned as well. Finally, ALEX is able to manage a multitude of different projects independently, each with their own set of symbols and learned models on a single workstation. Assume that there exist two or more versions of a

web application that differ in their feature sets. Then a project could be created for each version that can be managed without interfering with the others.

Another feature is the possibility of grouping symbols which allows an organized approach into modeling symbol sets. This means that a user can decide on how his symbols are managed. Thus, a task or a functionality-based grouping may increase the overview over a large amount of symbols. For example, a user could create a group whose purpose it is to hold symbols that define actions for resetting an SUL. As a consequence, they will not interfere visually with others. With the creation of the database of ALEX a permanent, non-deletable symbol group is created which contains unassigned symbols.

Symbols are identified by an unique identifier (id) that corresponds to a primary key in a database and a *revision* number. The attributes *group* and *project* contain the corresponding ids from the foreign key relation. They also have a *name* and an *abbreviation* which must differ from all other symbols of the same project. Abbreviations are displayed in hypotheses and limited in their amount of characters, whereas the name should be chosen meaningful. Furthermore, symbols can be moved across all existing groups.

With the creation and editing of symbols, a revision system is introduced. While managing actions or changing the name of a symbol, a new revision of it is created. This is done so, because when a learning process is started, the results will be saved with references to an id and revision pair from the learned symbols. This allows modifications of a symbol while maintaining existing learning results and makes it possible to undo mistakes in the creation phase by reverting a symbol to a previous version.

It can also be seen from figure 4.5 that symbols own a boolean *hidden* property. This flag determines whether a user executed a delete operation on it. In ALEX symbols are not permanently removed from the database because of the same reasons all revisions are saved. Instead, the possibility of making them visible again is given. As a consequence, restoring a symbol means to alter the flag. Contrariwise, symbol groups can be deleted permanently. Contained symbols would then be hidden and when restoring them, they are moved to the default group.

For interoperability and reuse of symbols across multiple projects or workstations, ALEX allows the export and import of symbols and associated actions in a JSON-encoded file. This enables to keep symbols that were created for a general purpose to be used even after the main project is not required any longer. Technically, users are able to manually create symbols and actions outside of the developed software, import them into a project and use them with all the benefits that the application provides. In order to prevent conflicts with existing symbol names, the name and abbreviation can be modified before an import.

Symbols own an ordered list of actions that are executed sequentially by the learner as soon as the symbol is called. Each one has a unique type attribute and represents an executable call to an SUL. Like the other objects, actions exist in a JSON format where

each JSON object is mapped to a parameterized piece of Java code as for example is shown in figure 4.6.

As figure 4.5 indicates, actions have the attributes *negated* and *ignoreOnFailure*. Typically, the outcome of an MQ is determined by the success of all actions or the fail of a single one. By setting the first one to *true*, the outcome of an action is negated in order to explicitly test if a condition is not fulfilled. The second one determines whether or not a complete word fails in case a single action fails. For example if the reset of an SUL starts with the logout of a current user in order to create a new one, it may be that he is already logged out. By setting the value to *true*, the failure is ignored and in this scenario, a new user can be created.

In the current version of ALEX the output alphabet $\Omega = \{OK, FAILED\}$ is used. On the one hand this makes it easier to skim the hypotheses and does not require any further configuration from a user. On the other hand, this might be too inflexible since it can not be determined which action caused the executed word to fail. For the front-end part, a possible future solution could be to add an additional property in every action object in which a custom error message could be stored.

The properties that are mentioned apply to all types of actions. In the following, different groups are presented that describe actions of logical togethernesses.

#### Web Actions

Actions of this group simulate real user interactions on the interface of a web application. The functions of these lean on the ones given by Selenium[12], a framework used for browser automating and testing purposes. It offers so called *web drivers* for various web browsers and methods for loading websites and executing actions on it. It is used on the server side of ALEX for testing reactions of an SUL to user input.

While Selenium offers a broad range of actions that cover a big part of user interactions, the possibilities of the application is limited to a subset that seemed relevant in the context of this work. Calling URLs, clicking on elements, filling out input fields and submitting forms are some of them. In order to let Selenium know which element to interact with, the CSS selector of it has to be given. In some cases it is wanted to check if a specific page is loaded or if a user interaction changed the content of the page. Therefore, actions that can search for the existence or absence of a string or an HTML element can be used. For more flexibility, regular expressions are supported, too. Moreover, it has been assumed that the effort for the implementation and testing of further actions is disproportionate to their usefulness for the target audience. This includes for example read, write and delete operations on cookies and the interaction with iframes and native dialogs.

[12] `http://www.seleniumhq.org/`

```
var actions = [
  { type: "web_fill",
    node: "#query",
    value: "automata_learning" },
  { type: "web_click",
    node: "#submit-btn" }
];
```

```
/** initialize webDriver **/
webDriver
  .findElement(By.id("query"))
  .sendKeys("automata_learning");
webDriver
  .findElement(By.id("submit-btn"))
  .click();
```

**Figure 4.6:** A list of actions in JavaScript (left) and their execution with Selenium in Java (right)

Figure 4.6 shows an example of how a JSON-representation of a list of actions would be implemented with Selenium in Java. The *webDriver* object is used for executing the methods on a real browser and had to be instantiated before.

A possible symbol for a web application could be to create a new user. Such a symbol could exemplary be realized by creating two actions for filling out input fields with an e-mail address and a password, followed by an action for submitting a form. The success of this interaction could then be verified by searching for the string *"User xxx created"* on the page.

Another problem when creating web actions that include the use of HTML elements is that the person who creates the symbol has to be familiar with HTML. In case several people have developed different parts of an application, the tester might not be the front-end developer. To give those users still a possibility to create these kind of actions, the selection of HTML elements has to be simplified as well. The concept of the *HTML element picker* provides that a person with little or no knowledge of HTML can extract a unique selector of an element by simply clicking on it. For this purpose, the application to be learned is loaded into and displayed in an iframe that is embedded in the front-end.

This intention comes with certain problems if an SUL is not accessible in the same origin. Let there be an iframe in a web browser and a server in origin $O_a$. If it is tried to load the contents of a website from origin $O_b$ into the iframe, the *same origin policy* that is implemented in web browsers prevents the loading due to security measures. A work around for this problem looks as follows: Instead of directly making a request to the website, a proxy is setup on the server in $O_a$ that is called with the requested URL. Then, the server loads the contents of the website and manipulates all references to resources in a way that they are also loaded over the proxy. The manipulated file is then send back to the web browser which can now display the requested site since the request lies in the same origin. As a result, is also makes the contents of the iframe accessible through JavaScript which in the end allows the selection of elements. This manner also enables to forward POST requests that for example contain cookies for authentication and session handling with a remote web application.

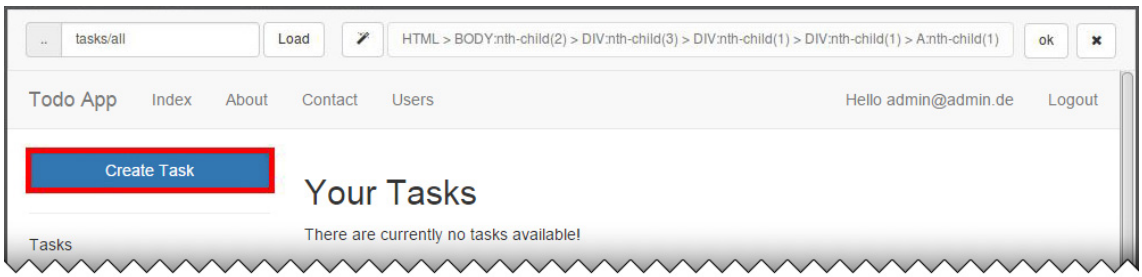


**Figure 4.7:** HTML element picker with a highlighted element and its selector in the header

As it can be seen in figure 4.7, selected elements are highlighted with a thick, red border and its CSS selector is previewed in the header. A click on the *ok*-button would automatically fill an input field for an action form with the selector. This especially simplifies the extraction of complex selectors that otherwise would have been entered manually, thus preventing typing errors. Furthermore, a user can navigate through the application as he can with normal browsers with the input element on the top left.

### REST Actions

The second group holds actions that are used for the interaction with RESTful APIs. The communication between an API and a client relies on sending and receiving HTTP requests and responses. So, the idea is to create actions that on the one hand can send parameterized HTTP requests to a URI and on the other hand can analyze HTTP responses. Contrary to web actions which function independently from each other, REST symbols should always start with a request that is followed by actions to analyze the response.

For sending HTTP requests to an SUL, one action allows to create a request with a method in {*GET,POST,PUT,DELETE*}, a URI and a content body that is send with it. Further methods like *PATCH* or *OPTIONS* are excluded because either it is not supported by jetty or they have been classified as less relevant for learning CRUD operations by the developers of ALEX after an evaluation.

HTTP responses can be examined for their status and an arbitrary header field. When for example checking if the server sends a JSON formatted resource, it could be looked up in the header if the value of the field *"file-type"* is *"application/json"* in order to decide whether an action and therefore the current word fails or succeeds. In most of the cases, it is expected that the received contents are formatted in JSON. So there are specific actions that check whether a JSON attribute exists or whether it has a specific value or type. While many RESTful APIs rely on this format, other formats like XML or unstructured data are only supported implicitly. A user always has the ability to search for a string or a regular expression in the HTTP body. For future releases of the application, special actions for this purpose may be added as well.

### General Actions

General actions have no logical connection with its web or REST counterparts but define methods that can be used by both and allow interaction between those groups. The idea behind that is that web applications and web services can be learned at the same time. In case that both parts share the same functionality, the deviation would be visible in the learned model, instead of checking the isomorphism of two graphs.

For the interoperability between symbols and actions, the possibility to create variables and counters via given actions are available. The action *Set Variable* expects a user to enter a name and a string value. If the variable did not exist it is implicitly initialized and kept alive for the current MQ. Counters are created with the action *Set Counter* and also require a name and an integer value. In contrast to variables, counters are persisted in a database with a reference to a project and can be incremented and reset as needed. This has the advantage that systems do not necessarily have to be reset manually in between different test runs. For example if the reset of an application is defined by the creation of a new user, a symbol could be created that increments an existing counter $i$ and then creates *user-i* like it has been done for learning ToDo in section 5.2.

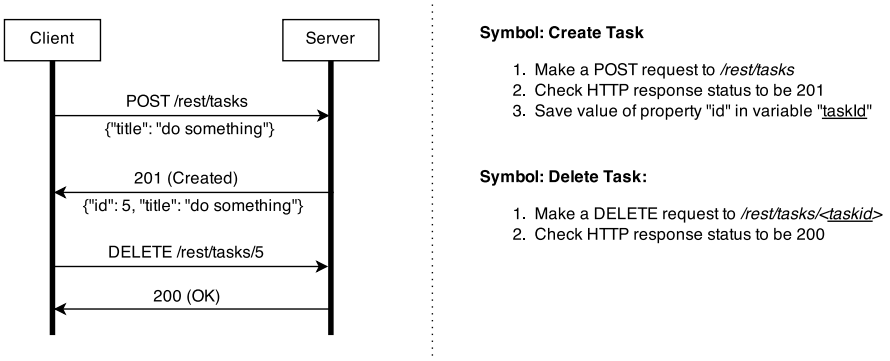


**Figure 4.8:** Possible interaction between client and server (left) and possible symbols (right)

Moreover, ALEX offers actions for dynamically filling variables by extracting a text value from an HTML element and a JSON attribute from an HTTP response. This is helpful in cases where new information is created during the execution of a symbol and required for another one. An example for this can be seen at figure 4.8.

| Type | Notation | Description |
|---|---|---|
| Variable | {{*$name*}} | Replaces the pattern with the value of the variable *name* that was created during the executed MQ. |
| Counter | {{*#name*}} | Replaces the pattern with the value of the counter *name* that is saved in the database. |

**Table 4.1:** Notation for variables and counters in actions

In order to use variables or counters in action properties, a notation is introduced where the possibility of a natural occurrence in a string sequence is very low. Table 4.1 lists both and summarizes their meanings.

A further action allows to execute another symbol. This can be described as follows: Let $s_1$ and $s_2$ be symbols with $s_1 := (a_1, a_2, a_3, \ ... \ , a_n)$ and $s_2 := (b_1, b_2, \ ... \ , b_m)$ where $a_i$ and $b_j$ are the list of actions the symbols consist of. Assuming that $a_2$ is the action that calls $s_2$, the resulting sequence of executed actions to an SUL would be $(a_1, b_1, b_2, \ ... \ , b_m, a_3, \ ... \ , a_n)$. This approach enables to build symbols that reuse the logic of other ones. As figure 4.9 shows, the logic of a reset symbol could be modeled by already existing symbols without lots of effort.

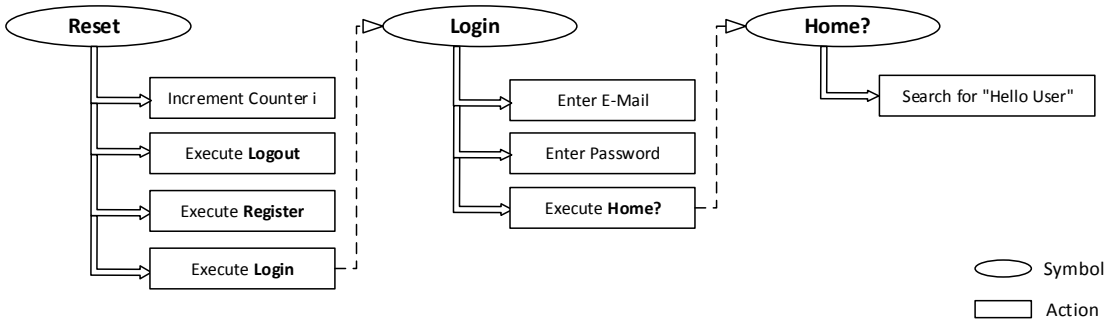


**Figure 4.9:** Possible reset symbol hierarchy

### 4.2.2 Learning Experiment Modeling

Contrary to LearnLib Studio, ALEX follows a form-based approach in modeling learning experiments. This is less flexible than the process-based one but it is a compromise in favor of a simple solution while trying to allow configuration to a certain degree. In addition, the goal of this phase is to keep the time it takes to create a learning experiment as low as possible. A disadvantage of ALEX at this point is that learning experiment setups can not be saved across multiple test runs. In upcoming versions the possibility of doing so is conceivable.

Let $S$ be the set of all created symbols with $s \in S$ is a symbol and $|S| = n$, then a learning configuration can be defined as follows:

**Definition 2 (Learning configuration)**
*A learning configuration is defined as a tuple C = (P,r,a,o,i) where*

- $P := \{(id_{s_0}, rev_{s_0}), (id_{s_1}, rev_{s_1}), ..., (id_{s_p}, rev_{s_p})\}$ *with* $s_1 \neq s_2 \neq ... \neq s_p$ *and* $1 \leq |P| \leq n$ *is a set of id and revision pairs from all symbols*
- $r := (id_{s_r}, rev_{s_r})$ *the id and revision pair of the reset symbol*
- $a \in \{Extensible\ L^*_M, DHC, Discrimination\ Tree, TTT\}$ *a learning algorithm*

- $o \in \{random\_word, complete, sample\}$ *an equivalence oracle*
- $i \in \mathbb{N}_0$ *the amount of maximum generated models*

The first step for a user would be to model the learning alphabet by selecting symbols or symbol groups from a list. The idea behind this is that a user is given the choice which parts of an application he wants to learn. This enables to quickly test small parts or single features of an application, like CRUD operations on a resource as well as learning the complete one.

A goal is to allow users who are not that familiar with active automata learning the execution of experiments with a minimal effort. As a result, a predefined learning configuration is set, but it can be parameterized in different ways. For once, the algorithm can be chosen. Available options are: Extensible $L^*_M$, a variant of the $L^*_M$ algorithm that is offered by the LearnLib, DHC, Discrimination Tree and the TTT algorithm which each differ in their internal data structures, performance and used space. For simplicity reasons, the Extensible $L^*_M$ is called $L^*_M$ in the following.

Then, a reset symbol $s_r$ has to be selected, whereas it can be included in the list of the learning symbols as well. As shown in chapter 4.2.1 a system reset can be modeled with existing actions in various ways. By offering a further approach to model a system reset, an additional and unnecessary layer of complexity would have been integrated which is not what ALEX aims for.

The next step would be to choose a combination of a learning algorithm and a parameterized equivalence oracle. For inexperienced users, this step is optional and a preset configuration is given. It can be selected from different algorithms and equivalence oracles that the LearnLib offers:

**Random Word**

This oracle creates random words from the input alphabet, executes them on the SUL and compares the result with the latest model. It can be parameterized with a value for the minimum and maximum length of generated words as well as with a value that defines the number of words per EQ.

**Complete**

The complete equivalence oracle creates words from all possible combinations of symbols, executes them on the SUL and checks the results against the hypothesis. The size of the words is limited by values for the minimum and maximum depth.

**Sample**

If this oracle is chosen, the learner stops with the first hypothesis and the browser displays it. Users are then able to search for counterexamples and verify them by themselves. They consist of a list of pairs of input and output symbols and can be sent to the server for the refinement of the hypothesis in the next iteration.

Finally, $i$ defines how many hypotheses should be generated by the learner before it is presented to a user. With $i = 0$ a hypothesis is only shown if the learner assumes that it has finished learning an SUL.

As soon as the configuration is set and a user initiates the learning process, a JSON-based representation of $C$ is send to the server that starts learning the application. Depending on the size of $i$ and the chosen equivalence oracle, the learning process is paused [25] either after a maximum of $i$ steps or after each step ($o == sample$). The generated model is then presented to a user. From that point on, he can adjust the values for $i$ and $o$, add counterexamples and induce the server to refine the hypothesis with the new configuration.

Directly aborting a running learning process is not possible due to untreated side-effects on the server. Instead, a user can initiate an abort-request which results in pausing the learning process after the current iteration. From there on, the configuration for resuming it can be changed as described.

### 4.2.3 Hypothesis-Based Interaction

Another feature that is implemented is the possibility to interact with the learned model. LearnLib supports the manual input of counterexamples while learning. So, the application should give users the capability of choosing some, too. In order to do so, two simple text input fields where a list of inputs and a list of expected outputs can be entered would suffice this goal. But this procedure can cause errors that can result to server-side failures by accidentally entering wrong symbol names or invalid outputs. The idea to minimize the error rate through mistyping is to select the inputs and outputs directly on the presented model. When the user clicks on an edge label, the output gets highlighted and a list of sortable input and output pairs is generated or expanded.

Counterexamples can but must not necessarily be validated before they are considered for the refinement of the model by sending them to the server. It then checks if the transferred sequence actually is a counterexample and sends a response with the outcome of the operation to the client. There, a user gets a visual feedback in form of a pop-up message that informs him about the status. After that, he can still decide whether to save the sequence for the next iteration or drop it and test further ones.

When learning large applications with many symbols, the generated models might get large and therefore hard to read. In large models there are typically states with many reflexive edges. It may then occur that the labels interfere with each other in the rendered model. So, the inputs and outputs are either hard to read or not recognizable at all. In order to enhance the situation, three solutions can be taken into consideration. The first one is to combine multiple reflexive edges into a single one and to combine all labels at the same time leading to a more clearly arranged model while keeping the same information.

Solution number two is to allow arranging nodes and edges of a graph manually. The last one is to set parameters to the layout algorithm which can be changed. Parameters such as the gap between reflexive edges and the distance between nodes may be helpful. In the end, a combination of all three would be optimal for a user. But since hypotheses are often re-rendered in the front-end the aspect of manual rearrangement is not implemented. This is because changed positions of nodes and edges are not taken into account for successive rendering processes.

Instead of static JPEG files, a more dynamic format is favored in order to maximize usability and reusability of generated models. The use of the SVG format promises advantages over pixel-based formats like JPEG. SVGs can be scaled without a loss in quality, animated and can be treated like HTML elements. They can also be extracted from web pages and be embedded in other documents, as it is the case in this work.

A further aspect to be considered is data continuity. If a user wants to follow the development of the learning process of its application over time he can do so. In a behavior that is known from image slide-shows, all intermediate models including the last created one can be clicked through in each step.

The side-by-side comparison of learning results (figure 5.6) forms another feature. This makes it possible to visually compare the results of multiple complete tests at the same time. So, for example a hypothesis and the corresponding internal data structure can be shown simultaneously. In terms of learning small parts of an application, multiple tests of the same set of symbols can be compared so that the results of changes that have been made during the development can be tracked better.

### 4.2.4 Representation of Internal Data Structures

Each learning algorithm uses certain data structures during the learning process from which a hypothesis is generated. For understanding how the algorithms work internally and how the model is built, ALEX supports the visualization of the data structures for the $L^*_M$ and the Discrimination Tree algorithm. Both are generated server-side by interacting directly with LearnLib. Finally, a string based representation is send to the front-end where it is parsed and presented beside its corresponding hypothesis. Since both algorithms have been introduced in chapter 2.5.1 and 2.5.2 the TTT and the DHC algorithm remain to be explained briefly.

The DHC algorithm as described in [28] creates and operates directly on the hypotheses while learning, so a representation of additional structures is not applicable.

An effort has been made by Schieweck [25] to create a JSON-based representation of the structure used by the TTT algorithm that can be seen in figure 4.10. Due to other priorities, this feature is delayed and planned for upcoming releases. Since the technology to render trees and hypotheses is already implemented, only a minimal extension of existing

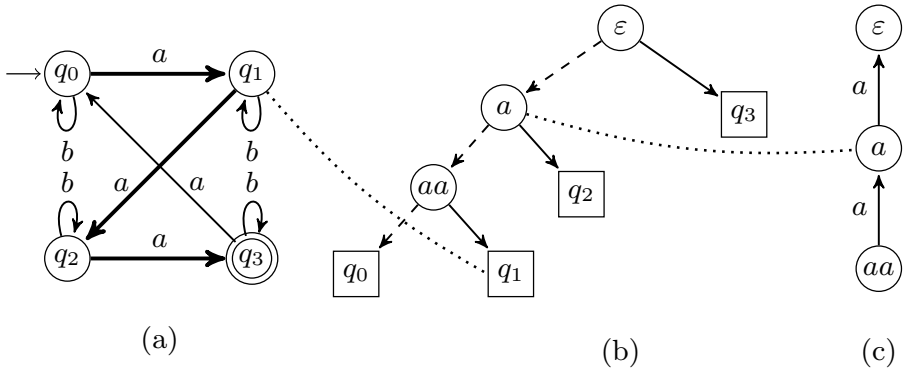


**Figure 4.10:** (a) target DFA and final hypothesis, (b) final discrimination tree, (c) discriminator trie for final hypothesis. Source: [14]

code would be necessary for drawing the structures and the association lines in the front-end.

### 4.2.5 Learning Result Analysis

Algorithms have different profiles regarding their runtime, used space, number of EQs, MQs and other values. The developed application is able to gather and visualize theses profiles in order to let users gain insight about advantages and disadvantages of using different algorithms in combination with differently parameterized equivalence oracles.

While learning an application, the server-side part of ALEX interacts with the LearnLib, whereat some numbers about current learning processes are gathered. While it is explained in [25] which values are extracted from the LearnLib and how it is done, this thesis roughly lists the gathered numbers.

For statistics, the number of *EQs*, *MQs*, *Called Symbols*, $|\Sigma|$ and the time it takes to generate each hypothesis model is saved. This means that users can learn about the characteristics of algorithms while having a chance to compare the generated results. For each iteration of the learning loop a user can have a look at the numbers that have been collected solely during the step and a cumulated version for all previous iterations.

The statistics module of ALEX presents the characteristics of test runs and lets users compare different profiles in line and area charts as described in section 4.3.6.

## 4.3 Implementation

This chapter emphasizes the implementation of the front-end of ALEX. For this matter, several approaches for realizing the concepts from chapter 4.2 are described, their intention explained, evaluated and finally the choice justified.

Instead of a classic, multi-page web site, the architecture of an SPA is used. This makes it feel like a native application and aims to make learning web applications quicker, therefore enhancing the workflow of a user in the hope of quicker results. The Interactivity, reactivity on a users actions and the easy management of application related objects is a main motivation to use web technologies. Moreover, some implemented features profit from this decision, since they would have been hard to be realized with languages like Java. As an example the HTML element picker can be named.

Furthermore, users should always know if their actions were successful or failed. Especially when a lot of data is sent to or received from a server, a user should be notified about server- or client-side errors or the outcome for actions. For this purpose, an application wide notification systems is applied. Notifications, that also called *"toasts"*, overlay the main page and display a temporary and removable text message. As an example, toasts with a red background inform about errors, green ones about successful actions and blue ones present useful information.

In section 4.2.1 the concept of a project has already been introduced as a way to manage multiple applications and corresponding resources independently. The front-end deals with the aspect of separation of resources in the following manner: When the application is opened for the first time, a list of created projects is presented and the only other accessible view is the one for creating new projects. Other parts that deal with resources of a specific project can not be displayed until a user "opens" one. In this case, a JSON object of the project is saved in the session storage of the browser. After that, other views like the one for editing symbols of the persisted project are accessible. A project can also be "closed" in a sense that it is removed from the session storage. This process results in the redirection to the list of available projects.

### 4.3.1 Used Design Patterns

The front-end of ALEX makes use of three common design patterns that are found in many web sites and in mobile, native and web applications: lists, modal windows and dialogs and dropdown menus. The implementation of those should make the developed application more familiar for users and at the same time simplify common tasks of automata learning. Each one is shortly introduced in the following.

#### Lists

The idea behind applying the list layout is that information is presented in a way users have grown accustomed to over time by using other web sites and mobile applications. As a consequence the work flow of the program should feel familiar, resulting in a lower barrier into active automata learning.

#### Modal windows/dialogs

Modal windows and dialogs offer a way of communication with the user, thus making

it more responsive. In many parts of the application users get asked to enter required information via prompt dialogs and can customize or parametrize other aspects in modal windows. They have the advantage of creating a clean user interface which only shows the necessary information while detailed things are hidden until a user's action reveals them.

**Dropdown menus**

Similar to the intention of using modal windows, drop-down menus can help to group logical items and only show them if they are needed.

ALEX is developed with an eye on mobile devices by making use of responsive design principles. This means that elements and contents of a page adjust to the browsers size by changing visible parts and rearranging elements. This is mainly achieved through the use of Bootstrap and CSS3 media queries [13].

### 4.3.2 The Interface

Instead of pages, the front-end is separated in *views*. A view is a part of a page whose content dynamically changes depending on users actions or the URL. For example, a view could display the list of all symbols under the URL *"/symbols"* and a list of actions under *"/symbols/{symbolId}/actions"*. The interface follows a continuous pattern. At the top of the page an omnipresent navigation bar is placed so that each feature is accessible with a minimum amount of clicks. Depending on whether a project is persisted or not, different navigation points are shown. If a project is not saved the number of accessible pages is limited to the one for creating new projects. On small screens, all navigation items are hidden and are accessible through an off-screen navigation that slides in and out of the screen as needed.

Below the navigation, the main view of the application is placed whose content changes depending on which navigation item a user clicks. Normally, a header with a title and a description of the current view is placed. The intention is to let users know where they are at any given time. On some views however, the use of it is disclaimed in order to have more space for other features. Then, its purpose is revealed through the previous interactions.

Under the header a sub-navigation holds buttons for different kind of interactions with objects of a view. It fixes itself below the main navigation when the viewpoint of the browser reaches a certain amount of pixels. So, the main navigation and the element for interaction are visible even if a long list of elements appear on a page that has to be scrolled through in order to reach elements outside of the browsers screen.

[13] `https://developer.mozilla.org/en-US/docs/Web/Guide/CSS/Media_queries`

### 4.3.3 Symbol Construction

The symbol construction is divided into the management of symbol groups and their symbols and the management of actions of a single symbol. Both tasks are handled on separated views.

#### Symbols and Symbol Groups

Symbol groups and their symbols are ordered in a tree-like structure, that is also known from file explorers. Symbols are displayed indented below their belonging group which can be collapsed as needed, so that the symbols are hidden temporarily. By using *angular-select-model*[14], items of a list are (multi-)selectable, which also resembles to the functionality of file explorers, so that batch operations on symbols are possible. The forms for creating and updating both, as well as moving symbols from one group to another are realized in modal windows. For each list item, a dropdown menu eases the access to these operations for a single item.

For restoring deleted symbols and older revisions two different views are installed that resemble each other in their logical structure. Each time, a list of symbols is displayed with a single possible action button for executing the corresponding operation. The only difference is that multiple symbols can be restored from the trash.

As mentioned, it is possible to im- and export created symbols from and into a JSON file. Thanks to HTML5 technologies, file downloads can be encapsulated from a server. The exporting process always proceeds in the same manner. First, data is gathered and encoded using the *encodeUriCompontent()* method of JavaScript. The resulting string is attached to the *href*-attribute of a link element and the *download*-attribute gets filled with the filename. The link is then appended to the Document Object Model (DOM) tree and a click on it is simulated which downloads the file. Finally, it is removed from the DOM tree in order to not pollute it. In this way, nearly every part of a website can be downloaded on the client side.

For the import, the generated JSON file can be dragged into the browser or selected via a file-dialog. Symbols are then listed instantly in the view. Users can again chose which symbols should be uploaded. This on the other hand brings problems when importing symbols into a project that already contains some because of possible naming conflicts. This is why the name and the abbreviation can be changed before the upload. Also, actions that call other symbols have to be manually adjusted in this scenario.

#### Actions

Actions of a single symbol are also presented in a list whose items can be sorted with drag and drop behavior for an easy rearrangement. The sorting functionality is implemented

[14] https://github.com/jtrussell/angular-selection-model

using the library *ng-sortable*[15]. The resulting order is then sequentially executed by the learner when it calls the symbol. Each action object has a unique *.toString()* method that states its task in a comprehensive manner. Under each description, labels are displayed that show if the result of an action is negated or its failure is ignored. As it is the case with symbols, multiple selection of and batch functions on them are allowed for a reduced effort.

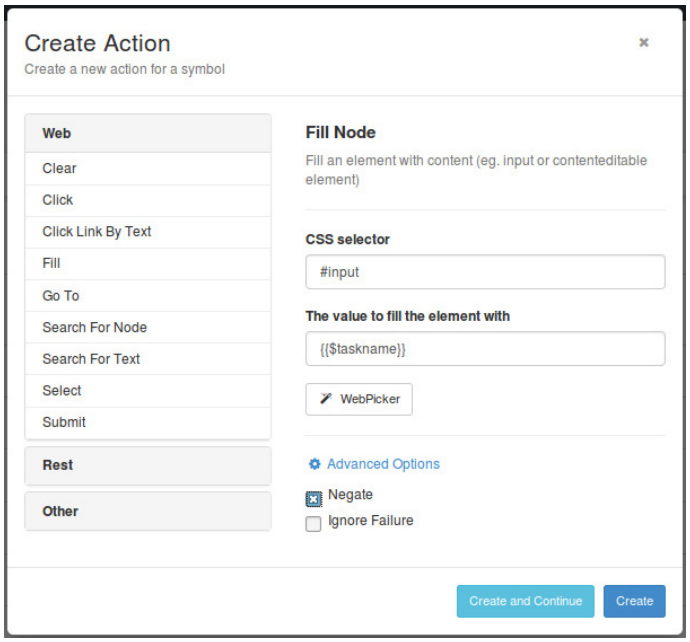


**Figure 4.11:** Creation of an action in the action editor

The creation and editing of actions are parts of the view that are realized in modal windows as it can be seen in figure 4.11. Due to the amount actions that given by the editor, the use of them prevents the cluttering up of the interface. The window for the creation is separated into two columns, where on the left side, an accordion menu shows the action groups with their available actions. On the right, the corresponding form for the selected action type is shown. What can also be seen in this example is that the HTML element picker is embedded in supported forms as a button. A click on this button opens the picker and lets users interact with the web application. The selector of a selected element appears automatically in the first input field.

Many web actions require users to enter a unique CSS selector of an HTML element. A button on these actions opens the HTML element picker that is described in chapter 4.2.1 and is shown in figure 4.7. It lays above the interface like a modal window, but nearly reaches the dimensions of the whole browser. With a text input for a URL, it functions like a reduced web browser. When the page is loaded for the first time, the

[15]https://github.com/a5hik/ng-sortable

page behind the base URL of the project is loaded into the iframe. Then on each visible link a click event is attached on which the value of the *href*-attribute is extracted and saved. So, when the HTML element picker is closed and reopened, the last visited page is loaded. This brings an advantage for example when creating actions that fill out forms with many text inputs. So, instead of going to the required page for every input from the base URL, a quick access is given. The HTML element picker has two different modes. In the browsing-mode, users can interact with the loaded page as they would in other web browsers. If the *selection*-mode is enabled, elements that the user hovers over are decorated with a red border. At the same time, the unique CSS selector to this element is presented in the header. A click then ends the selection mode and saves the selector as well as the *.textContent* value of the last visited element. During the click, it is tried to prevent other click events to happen that for example redirect to another page. In a factory object, these values are available for other components of the application to use.

For sending HTTP requests with *PUT* and *POST* methods the embeddable code editor *ace*[16] an the wrapper *ui-ace*[17] for AnglarJS were used in favor of a textarea element. It allows syntax highlighting, syntax checking and line indentation, therefore making it possible for a user to correct its errors. In the current version of ALEX, JSON is the only supported format but any other text can be entered as well.

At each given time a user can save the current state of a symbol and revert changes to the last update. As soon as a user modifies a symbol, an info message is displayed that informs about unsaved changes. This is necessary, because updating a symbol for every action would result in the creation of new, redundant revisions.

### 4.3.4 Learning Experiment Modeling

Before a learning process can be started, it has to be configured. Therefore, another view is provided. It has the same logical organization as the one for managing symbols and symbol groups and shows a similar system for using it. Symbols that should form the alphabet can be selected as well as the symbol that should be used to reset the application.

An info panel above the list always shows the current configuration, containing the used algorithm, the equivalence query and its parameters, the reset symbol and the maximum number of models to generate during the test run. Optionally, these parameters can be changed in a modal window that can be seen in figure 4.12. It has been discussed whether the reset symbol should also be included in the modal window. In the end, the selection of a reset symbol is obligatory and since every symbol can function as one, it seemed more natural to list them among all others. A possible learning experiment for learning CRUD on to-dos with the default configuration is presented in figure 4.12.

[16] http://ace.c9.io/
[17] https://github.com/angular-ui/ui-ace

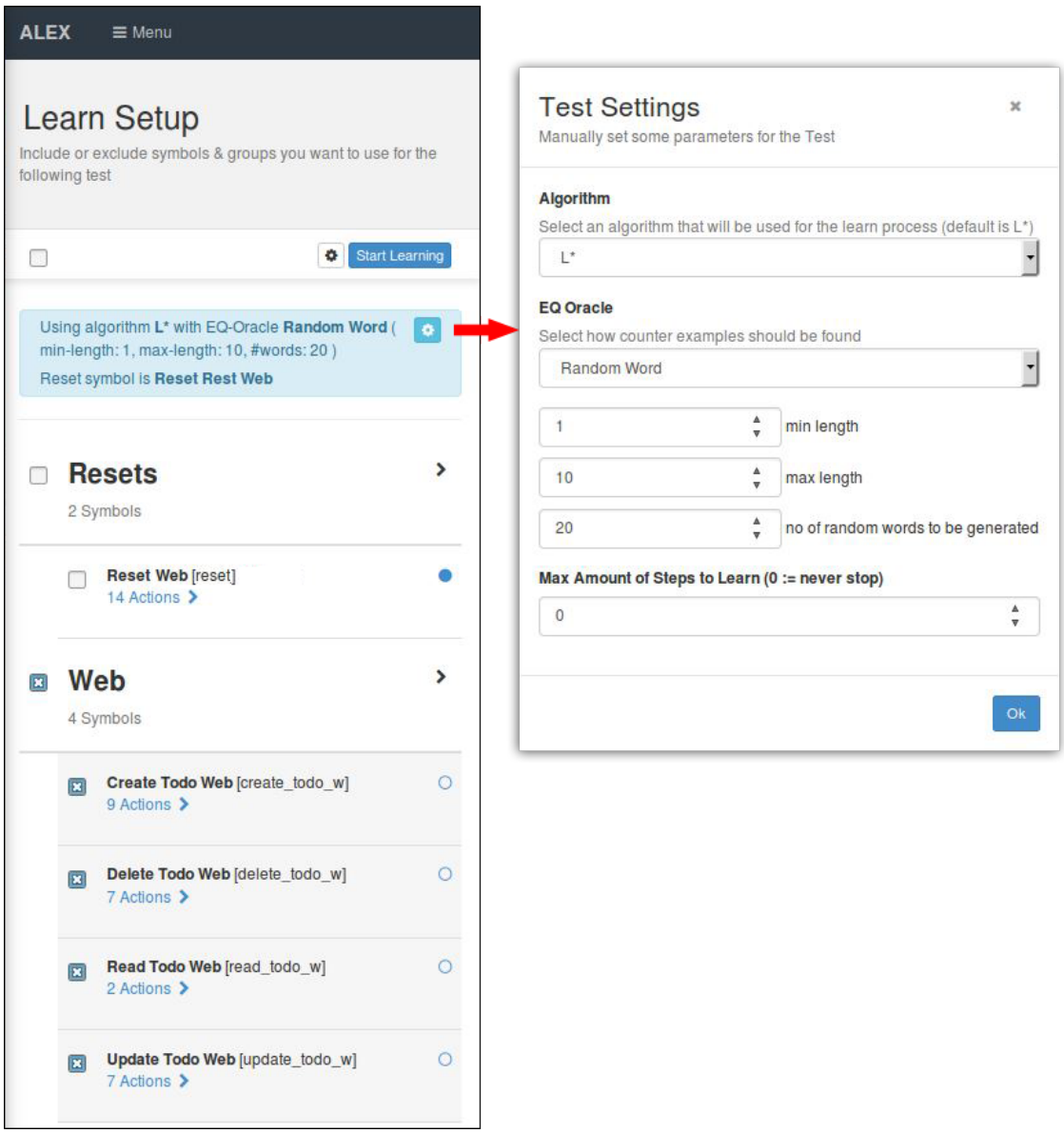


**Figure 4.12:** Possible learning experiment setup

As soon as the configuration phase is finished, the setup is send to the server that analyzes the received data. If the configuration is not valid, a message is send back to the front-end that displays the error message in a toast. In contrast to that, a success message is displayed and it is automatically redirected to an animated load screen where results are displayed as soon as the learner finished the generation of a model.

For getting the status of the learner, several information receiving methods were evaluated that are listed below.

### HTTP Pull

This technique describes a polling mechanism using AJAX, whereat requests are

send to a server at a user-specified, static or adaptive interval [9]. As a consequence, network traffic may increase due to possibly unnecessary server calls.

**Comet**

Comet is a protocol that is also based on AJAX requests. It allows a server to send messages to a client by keeping a so called *Long Polling* connection alive for a specified time [9]. During that, events occur that induce the server to send data to the client that then starts a new request. The same goes for timeouts.

**WebSockets**

WebSockets are described in the HTML5 specification and offer a persistent, bidirectional communication between client and server over TCP [18]. Thus, it reduces the network traffic that is caused when using the solutions above.

Finally, the *HTTP Pull* method is used due to the concepts of a stateless web service. For the second point, the server would have to keep a connection alive over a period of time instead of responding as fast as possible in order to free resources for other requests. The last point contradicts to the idea of a stateless interface in a sense that the socket is created permanently for the communication with a client and the server would contact a client actively. On a local machine, where both, client and server are installed, the aspect of high network traffic usage can be neglected. Moreover, the generation of learned models can, in some cases, be a time consuming process. So, the refresh interval is set to a value where results from fast learning processes can be displayed with a justifiable delay.

Once the load screen is shown and the learner is active, some restrictions in using ALEX are made. A return to the learning configuration view from the current or another project is not possible because it is not allowed to start more than one processes. Also, updating or deleting projects is not possible because the learner depends on the base URL of the project.

After clicking the button to abort the current learning process, a feedback is given by an info message that tells the user that the learner pauses at the next possible moment.

### 4.3.5 Hypothesis-Based Interaction and Internal Data Structures

For rendering hypotheses models in a web browser, several libraries have been taken into account. One of them is *Cytoscape.js*[18], a network visualization tool using the HTML5 Canvas technology. It can also be used to render Mealy automata with little configuration and has been used in early development stages. Similar to this, *vis.js*[19] has also been considered. Finally, a solution using a combination of D3 and DagreD3[20] is used. The last one calculates the positions of nodes and edges of a graph and lays it out using D3. The use

[18] http://cytoscape.github.io/cytoscape.js/
[19] http://visjs.org/
[20] https://github.com/cpettitt/dagre-d3

of the SVG technology has some advantages in this use case. Firstly, they are embeddable in the default HTML document and therefore offer to attach JavaScript events to single elements which is required for selecting counterexamples from it. Secondly, SVG files can be exported into a scalable file which can then be manipulated outside of ALEX and can be embedded in other applications or documents. This increases the reuse of the format in contrast to static JPEG files that can be created from canvas elements.

ALEX offers several ways to interact with hypotheses. The first place to do so is during the learning process. On demand, each generated intermediate hypothesis of a test run is presented and the process stopped for the moment. The model is then rendered on the full screen for a maximal overview. Zoom and drag events on the SVG element are more interactive and easier to observe. Also, the initial node is highlighted to create a contrast from the other states. A pagination element makes it possible to click through all previously generated models of a test, therefore allowing to observe the progress the learner makes over multiple learning steps.

Known applications like LearnLib Studio make it possible to manually rearrange nodes and edges with a drag-and-drop behavior for optically structuring the learned model. This feature has not been adopted in ALEX, because it brought several inconveniences with it. The rearrangement can be a time consuming process and updated positions are not saved. This makes it only temporal change that is reverted as soon as the model is rendered again. In order to compensate this, a modal window is available where parameters for the layout can be specified, naming for example the distance between edges and nodes.

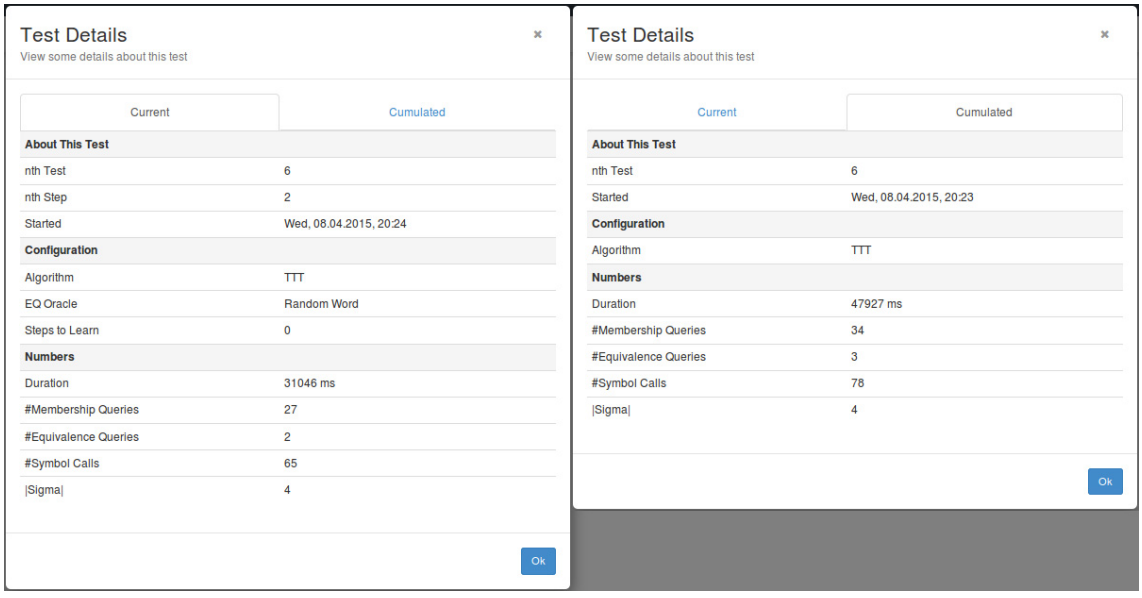


**Figure 4.13:** Test details for the current iteration (left) and a cumulated version for all previous ones (right)

Other features include the export of the presented hypothesis as SVG and JSON, as well as another modal window that allows to view mentioned statistical values that are gathered during the test as shown in figure 4.13.

Another point is the representation of internal data structures. As stated in chapter 4.2.4, the presentation of the observation table of the $L_M^*$ and the discrimination tree of the corresponding algorithm is possible. This is realized in the same view where the model is rendered. With the click on a button, the current visible model is replaced with the internal data structure of the algorithm. Another click switches the view back again. For the observation table, its ascii representation of it, that is fetched from the LearnLib, is parsed and converted into an HTML table. On demand, it can be exported as a CSV file for later observations and integration in other documents. The same behavior is implemented for the discrimination tree. Its tree representation that is received from the API is therefore converted into a graph structure, so that it can be rendered with the same libraries as the hypotheses. This way, the additional integration of a further library is avoided while again having the possibility to export it as SVG. The explained pagination behavior works in this viewing mode, too.

When continuing the learning process two options are given. The first one is to resume the process with the same configuration it started with. For the advanced usage of ALEX, the resume configuration can be changed if required. For this matter, a sidebar can be made visible that contains multiple widgets that can be used for the configuration. It only takes little space on the side of the screen, so that the observation and interaction with the displayed model is still possible during the configuration. On the one hand the customization of the used equivalence oracle for the next learning iteration is allowed. Despite that, counterexamples can be entered manually in case the chosen oracle is the *sample* one. Therefore, a sequence of input and output symbols has to be constructed by clicking on the edge labels of the last generated hypothesis. Each pair is then listed in the sidebar. They can be rearranged in its order via drag-and-drop and the output symbol can be switched from *OK* to *FAILED* and vice versa by clicking on it. For entered sequences, it can be tested if it really is a counterexample or not. It is send to the server which checks it and the result of this procedure is displayed in a toast message. When the configuration is done, the learning process can be resumed.

The second possibility to interact with hypotheses is during the post learning phase. On a separate view users can put multiple hypothesis panels of different or same learning results side by side, each allowing the abilities that were mentioned above, except the selection of counter examples. With this feature it is possible to have a look at for example a hypothesis and its internal data structure at the same time. Another possibility is to have a step by step comparison of two or more results from a feature that has been learned twice or more in order to observe changes. The number of simultaneously displayed panels is only limited by the browsers screen width.

Finally, it can be mentioned that both views are currently not optimized for mobile devices, yet it is planned to support smaller display as well in next versions of ALEX, for example by integrating touch gestures for zooming into models or selecting counterexamples.

### 4.3.6 Learning Result Analysis

In the context of the previous section 4.3.5 a modal window is presented that shows statistics of the number of conducted membership and equivalence queries, the duration of a test and other characteristics as well for a single generated model. In order to extend this idea, a statistics view is implemented in the application.

The analysis of learning results is limited to displaying statistics that were gathered during the learning in form of diagrams. For this purpose some libraries were considered that focus on drawing diagrams. The first one is *Chart.js*[21]. It makes use of the HTML5 Canvas technology and supports, amongst others, drawing line and bar charts. Again, the problem when exporting those is that the files are either in a JPEG or PNG format, which is not ideal for reusing them. It has been experimented with *canvas2svg*[22], a library that promises to mimic a JavaScript canvas object and transfer native calls into commands to create a SVG element that looks exactly like the original canvas. Due to incompatibility or insufficient support of draw functions of *chanvas2svg*, this solution could not be realized. Instead, the charting feature is implemented with *n3-charts*.

The statistics view of ALEX is used to visualize the values that are mentioned in chapter 4.2.5. Therefore, users are confronted with a list of all test runs of the opened project from which they can select multiple learning results in order to compare them. When the selection is made, it can either be displayed a bar chart of only the final results or an area chart, where the numbers from all intermediate steps of the selected results are simultaneously presented. In the first case (figure 4.14), the x-axis is labeled with the number of the selected test runs whereas on the y-axis the value of the selected characteristic is shown. In the second case (figure 4.15) the x-axis is labeled from 1 to $n$, where $n$ is the maximum number of intermediate steps from all learning results to compare. During the preparation phase, zero values are created for data sets of the tests where $|intermediate\ steps| < n$, so that they can also be displayed. Another advantage of *n3-charts* over *Chart.js* is shown in the visualization of charts. A legend below the diagram lets users change the visibility of single or multiple data sets. Furthermore, a button group that is located below the chart lets users switch the displayed characteristics without refreshing the page.

As default, the size of the chart is vertically limited for a better visual integration into the structure of the site. When comparing large records, this space might not be enough

[21] `http://www.chartjs.org`
[22] `https://github.com/gliffy/canvas2svg`

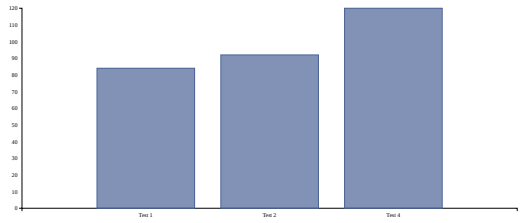

**Figure 4.14:** Statistic for multiple final learning results

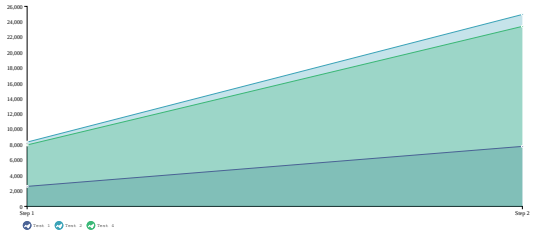

**Figure 4.15:** Statistic for multiple complete learning processes

for displaying all values in a readable way. That is the reason why the possibility to stretch the diagram to the full width of the browser is given, thus allowing a more precise look on the charts.

With the same method that is used to download hypotheses models and discrimination trees, the generated diagrams can be exported into an SVG file. Where the statistics page of ALEX comes to its limits, third party applications might be better suited for analyzing huge amounts of data. For this purpose, the statistics are gathered and can be downloaded as a CSV file, either for multiple final or a single complete learning result.

# 5 Evaluation: ToDo

As a part of this thesis the functionalities of ALEX are tested to ensure its usefulness for learning medium-sized web applications. Therefore, a web application and a web service are learned, the results analyzed and the usefulness of the developed solution evaluated.

The case study this chapter is about is *ToDo* (figure 5.1), the project that is described in the motivation for managing to-dos. It has been developed in Java using the web application framework *Tapestry*[1]. ToDo is designed as a multi user application with a role management system that separates users in administrators and default users. To its core functionalities belong the execution of CRUD operations on to-do objects with the ability to categorize them and to assign priorities to them. Beside the main application a RESTful service that allows to manage to-dos via an HTTP interface has been developed, too. For the protection of user specific resources, an API-key system is implemented. API-keys are random but uniquely generated strings of a fixed length that users can only create in the front-end. They have to be passed in each URI to get access to a resource, as in *GET http://localhost:8080/rest/{apiKey}/tasks/**. Although this method is not the most secure one, it sufficed the requirements.

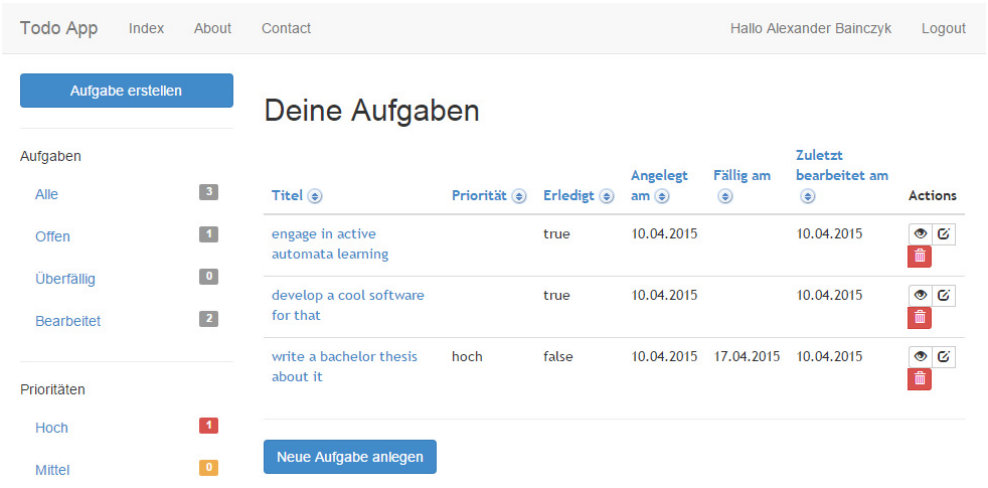


**Figure 5.1:** Interface of ToDo

[1] `http://tapestry.apache.org/`

ToDo has been learned with the LearnLib by manually creating a preconfigured learning loop, hard-coding symbols using Selenium for interactions with the web interface and classes from the Java EE *javax.ws.rs* package for the API.

## 5.1 LearnLib, LearnLib Studio, ALEX – A Comparison

In the following sections, an exemplary comparison between the use of LearnLib, LearnLib Studio and ALEX in terms of setting up a simple learning experiment for ToDo is given. For each tool and listed aspect, a similar situation is established, so that they can be compared in terms of flexibility, simplicity and effectiveness.

### 5.1.1 Symbol Construction

For the purpose of demonstrating the construction of symbols, a selenium symbol is modeled for creating a new to-do object with all three tools. The logic of the symbol is the following: first, a link is clicked that redirects to the page with a form for creating a to-do. Then, the input element for the title is filled. After that, a button for submitting the form is clicked and finally, a success message is looked up in the HTML document.

```
public class CreateTodo implements ContextExecutableInput<String, WebDriver> {

  @Override
  public String execute(WebDriver webDriver) throws SULException, Exception {
    try {
      By selector = By.ByCssSelector("body > div.page.container > div > div.sidebar.col-md-3 > a");
      webDriver.findElement(selector).click();
      webDriver.findElement(By.id("title")).sendKeys("Test Todo");
      webDriver.findElement(By.ByCssSelector("#task > div:nth-child(7) > div > button")).click();
      webDriver.getPageSource().contains("'Test Todo' erfolgreich erstellt!");
      return "OK";
    } catch(Exception e) {
      return "FAILED";
    }
  }
}
```

**Figure 5.2:** Symbol modeling with LearnLib

Figure 5.2 presents the manual approach with the LearnLib, figure 5.3 the process-oriented method of the LearnLib Studio and figure 5.4 shows the way ALEX deals with symbols. As it can be seen, all solutions have different approaches.

On a source code level by using the LearnLib, the possibilities for modeling a symbol are only limited by the functions of the libraries that are used. This might be an advantage for experienced developers, but people with little to no technical background may not come to satisfying results. Furthermore, the flexibility suffers from the hard-coding. In case there are changes in an HTML template that affect the symbol, the code has to be altered each time manually. In 16 lines of code, the symbol can be realized. If the amount and complexity of symbols rises, the code may be hard to manage.

LearnLib Studio requires less technical background but it is also only as flexible as the range of available SIBs that are given. Here, eight Selenium-based SIBs are used

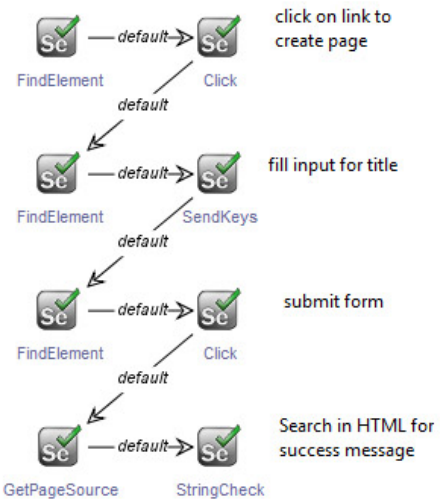


**Figure 5.3:** Symbol modeling with LearnLib Studio

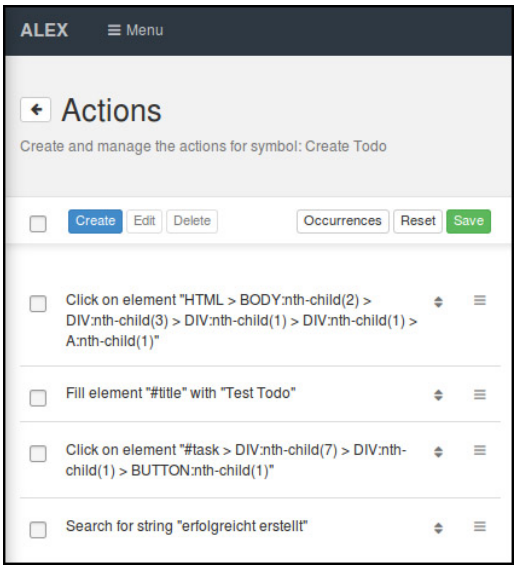


**Figure 5.4:** Symbol modeling with ALEX

to model the logic of the symbol. Again, selectors of elements have to be entered by hand. Furthermore, changes in a model are automatically saved in a new file which can be recovered by renaming it manually. Another advantage of this solution that is not presented in this example, is that conditional data flow can be modeled with different kind of edges that resemble *if ... else ...* control structures so that errors can be handled. For changing the order of SIBs, the structure of the model has to be changed by redirecting edges.

ALEX is the least flexible solution, since it offers a limited set of available Selenium actions and conditions can only modeled implicitly using the action's properties *negated* and *ignoreOnFailure*. In return, symbols can be modeled and actions resorted time-efficiently. In this example, the symbol for creating a to-do is created by four actions that correspond to the code lines in the *try*-block in figure 5.2 and to every SIB-pair in figure 5.3. Furthermore, changes of a symbol are always saved and can be reverted without the need to manipulate files.

Finally, the use of ALEX achieves the quickest results and the modeling takes less time compared to the other tools, especially because the selectors are generated semi-automatically with the HTML element picker.

### 5.1.2 Learning Experiment Modeling

Now, the learning setup from figure 4.12 is remodeled with LearnLib and LearnLib Studio. This means that an alphabet of length four is learned with the $L_M^*$ algorithm and the random word equivalence oracle. Finally, the observation table and the final hypothesis is presented.

The creation of learning experiments by implementing functions of the LearnLib directly turns out to be non-trivial process. The alphabet has to be hard-coded, as well as the learning-loop, the algorithm and the oracle. Learning experiments with different combinations of alphabets, oracles and algorithms are only circumstantial to realize by hand. Because of the extent of the preparatory work, a code snippet for the learning setup of the current example is not given at this point, but briefly explained. This can be done by creating two classes: a *TodoLearner* class that implements the learning loop and contains configuration for algorithm, oracle and alphabet. Then, a *TodoDataMapper* that implements the *Mapper* interface of the LearnLib where the reset mechanism of the application is implemented, symbols are mapped to concrete input strings and that returns a *WebDriver* for connecting the learner with the concrete SUL.

The presentation of an observation table can be realized by a single line after the learning loop. It creates a new HTML file and opens it with the default application of the host system:

```
OTUtils.displayHTMLInBrowser(learner.getObservationTable());
```

For the presentation of a hypothesis it is required to have the library *Graphiz*[2] installed on the system for exporting models into image or vector files. This results in an additional effort and a possible error-prone solution in case there are compatibility issues between the library and the system.

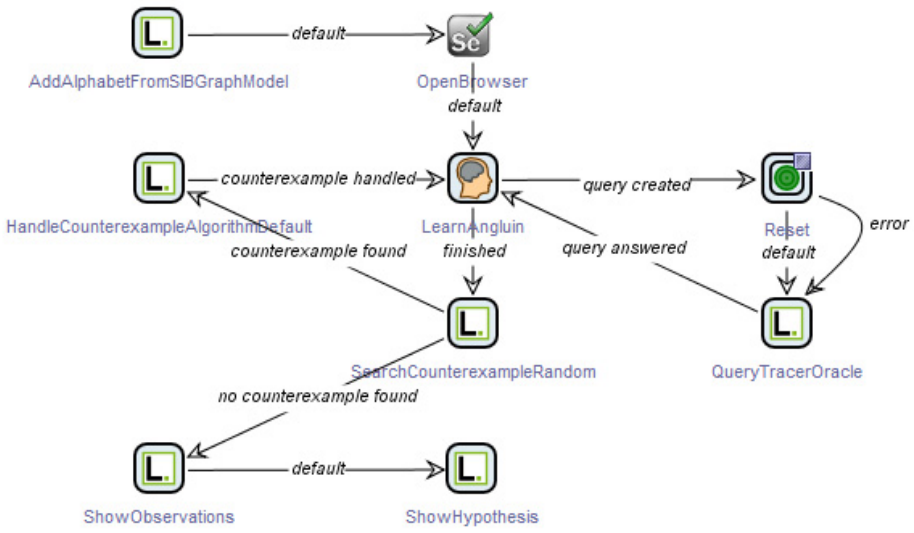


**Figure 5.5:** Possible learning experiment with LearnLib Studio

LearnLib Studio simplifies this process with SIBs that can be replaced and parameterized at will. The presented model in figure 5.5 can be compared with the learning setup that is shown in figure 4.12. The two SIBs in the first row can be compared to the selection of symbols from the symbol list. Contrary to ALEX, a new graph model for each

[2] `http://www.graphviz.org/`

alphabet has to be created in LearnLib Studio. The learning loop that is modeled in the two rows in the center corresponds to the settings that can be made in the modal window. Except for the reset symbol, that has to be selected, too. The two SIBs in the last row for displaying an observation table and the hypothesis are realized automatically with ALEX. The way how both results are presented in ALEX and LearnLib Studio are alike. Instead of an ability to toggle between internal data structure and hypotheses, LearnLib Studio uses tabs for presentation. Moreover, both tools do not rely on third party libraries.

As a conclusion, the preparation for a learning experiment and its customization takes the least time and effort in ALEX compared to a similar setup in LearnLib or LearnLib Studio in this case. However, it is to mention that ALEX does not save the configuration for learning processes. So, a successive execution of the same experiment takes more time than with LearnLib or LearnLib Studio, since the alphabet and the reset symbol have to be selected each time.

## 5.2 Learning ToDo with ALEX

In this section, the preparation and execution of a learning experiment for ToDo using ALEX is presented. For the purpose of readability of the generated hypothesis, only a subset of the presented features of ToDo is learned. The created symbols are limited to modeling CRUD operations on to-dos via the web and the REST interface, as well as the user login and logout. The following list shows the abbreviations and meanings of the symbols that are shown in figure 5.6 and 5.7.

**login_w** Logs a user in the system
**logout_w** Logs a user out the system
**create_todo_w** Creates a new to-do via the **w**eb interface
**read_todo_w** View a to-do via the **w**eb interface
**update_todo_w** Updates a to-do via the **w**eb interface
**delete_todo_w** Deletes a to-do via the **w**eb interface
**create_todo_r** Creates a new to-do via the **R**EST interface
**read_todo_r** Fetch a to-do via the **R**EST interface
**update_todo_r** Updates a to-do via the **R**EST interface
**delete_todo_r** Deletes a to-do via the **R**EST interface
**reset** Creates a new user and an API-key

As an example, the logic of symbol *reset_rest* is explained in detail. Note that ToDo originally does not have an interface for clearing the database. So, the reset is performed through the creation of a new user with an unique e-mail address.

At first, a potential logged in user is logged out from the system. Then, a counter $i$ is incremented or implicitly created on the first reset. After that, the variables *taskid*,

*taskname* and *apikey* are initialized with an empty value. It follows the creation of a user with the e-mail "user-$i$@user.de" via the provided form on the start page. For generating an API-key, the user is logged in, the page for creating keys is called, the provided form filled and submitted. Then, the value of the API-key is extracted from the HTML page and saved into the variable *apikey*. Finally, the front-page is called. As it can be seen, the described symbol can function as a reset in both scenarios.

With each MQ, the variable *taskid* is dynamically filled during the creation of a task by extracting the value from the HTML page. It is then used by REST symbols, for example for deleting a to-do object by sending a request to *DELETE /rest/*{*apikey*}*/tasks/*{*taskid*}. The variable *taskname* is filled with the name of the created or updated to-do name and used to test whether or not the object still exists.

For the first experiment, the three different alphabets are tested one after another for checking the equivalence of CRUD operations on to-dos for web and REST symbols:

- $A_1 := \{create\_todo\_w, read\_todo\_w, update\_todo\_w, delete\_todo\_w\}$
- $A_2 := \{create\_todo\_r, read\_todo\_r, update\_todo\_r, delete\_todo\_r\}$
- $A_3 := a_1 \cup a_2$

As algorithm, the TTT is used in combination with the random equivalence oracle. The results that can be seen in figure 5.6 show two things. Firstly, that ToDo handles CRUD operations on to-dos the same via both interfaces and secondly, that ALEX is able to learn both symbol sets at the same time.

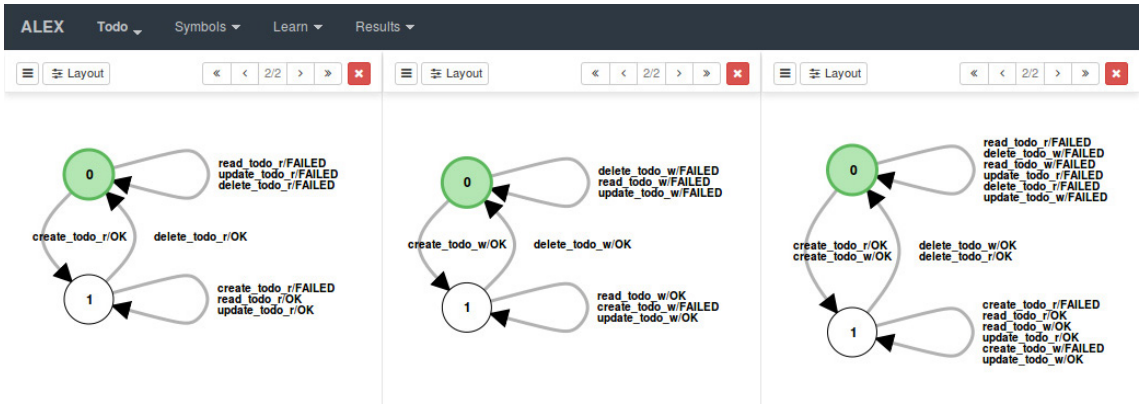


**Figure 5.6:** Visual comparison of different learning results

The second experiment only validates the correct behavior of the web interface. This time, the alphabet to be learned is $A_4 := A_1 \cup \{login\_w, logout\_w\}$. As a reset symbol, a slightly modified version of the original one is used that logs a user out at the end, so that the login and logout behavior is learned as well. The learning configuration stays the same as in the first experiment. The final hypothesis that is presented in figure 5.7 shows that ToDo functions as intended.

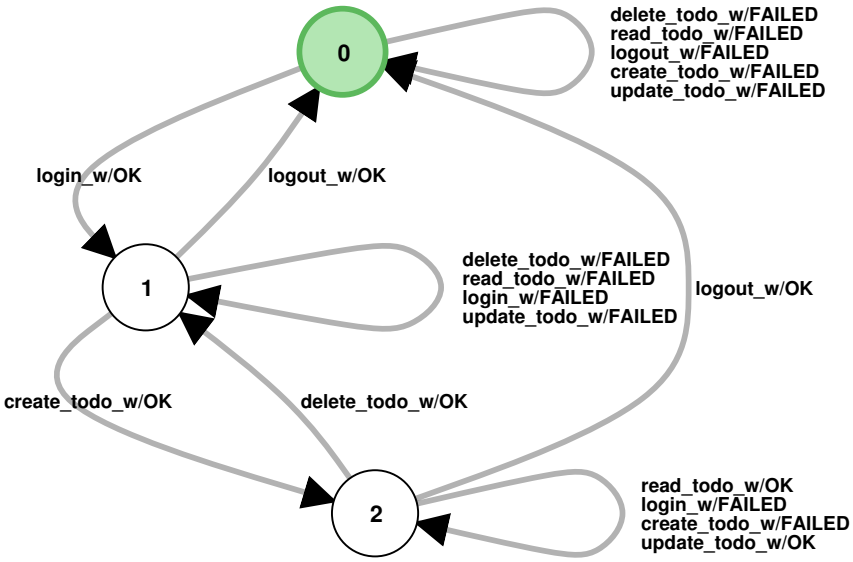


**Figure 5.7:** Final hypothesis of ToDo

# 6 Conclusion

The concepts that are described in this document for a simplicity-oriented, web-based approach to active automata learning of web applications and web services show that a manual learning setup with the LearnLib can be avoided in both cases. Based on the results of chapter 5 it can also be noted that ALEX can be more time efficient and easier to handle than LearnLib and LearnLib Studio in comparable situations.

The use of the developed graphical interface that is introduced and explained in the context of chapter 4.2 offers users a flexible alternative for creating symbols and modeling learning experiments. Especially the possibility to quickly build symbols by connecting atomic actions reduces the effort of a manual creation. In combination with the HTML element picker that is presented in section 4.2.1 for creating Selenium-based symbols, the amount of required technical background knowledge in order to learn web applications is reduced. Another success that can be recorded from the standpoint of usability is the simplification of the interaction with hypotheses. The easy selection and validation of manually entered counterexamples directly on a model allows a less error-prone approach as comparable solutions, even for more experienced users. Also, the possibility to observe the different stages of a model over a learning process proves to be helpful.

Furthermore, the practical appliance of the software on the case study in section 5.2 shows that ALEX can be used for learning web applications and web services while producing valid results. Therefore, the study of its usefulness by applying it to further web applications is justified.

Finally, the separation of the software in a RESTful API and a web-based client turns out to have certain advantages. The back-end handles the persistence of data and interacts directly with the LearnLib, so that resource intense operations can always be executed on a central and powerful machine. In addition to that, all functionalities are accessible via a network. The web-based client however can be used from a variety of Internet-connected devices, regardless of their power and make use of the API for the LearnLib.

## 6.1 Future Work

In the course of this work several points are mentioned that leave space for the extension of existing and the development of further features of ALEX. One example is the fixed set of available actions that may not cover all possible interactions with web applications.

An idea is to create a plug-in system that allows to add third party actions to the current set. A plug-in could consist of an HTML template for a form, a JavaScript file and a Java class with front-end, respectively back-end logic of an action. This could potentially also lead to the expansion of target use cases to various applications instead of just web-based ones.

Then, the visual feedback while a learning process is active is minimalistic at the moment and is reduced to showing a load-screen. Further approaches could provide a more informative presentation of details of the current process. This could for example be realized by presenting live data from the characteristic numbers that are described in section 4.2.5. If the idea of live visualization is developed a step further one could imagine to present the composition of the hypotheses, which would make sense if algorithms like DHC are used.

Another aspect is the extension of possibilities for modeling learning experiments for advanced users. With the existence of the WebABC a code base for a web-based process modeling tool is already available. So, parts of it could be integrated into a separate view while keeping the current setup. This would also make changes in the API design of ALEX necessary.

Thanks to existing frameworks like PhoneGap, the interface of ALEX could be ported to mobile platforms as Android and IOS, thus making active automata learning more mobile friendly and accessible. This would require to make the current interface more touch-optimized by integrating further libraries that specialize in interpreting touch gestures.

# Table of Figures

# Table of Acronyms

**AJAX** Asynchronous JavaScript and XML

**ALEX** Automata Learning Experience

**API** Application Programming Interface

**CORS** Cross-Origin Resource Sharing

**CRUD** Create, Read, Update, Delete

**CSS** Cascading Style Sheets

**DFA** Deterministic Finite Automata

**DOM** Document Object Model

**EQ** Equivalence Query

**HTML** Hypertext Markup Language

**HTTP** Hypertext Transfer Protocol

**HTTPS** HTTP Secure

**JSON** JavaScript Object Notation

**MQ** Membership Query

**REST** Representational State Transfer

**SIB** Service-Independent Building Block

**SPA** Single-Page Application

**SUL** System Under Learning

**SVG** Scalable Vector Graphics

**URI** Uniform Resource Identifier

**URL** Uniform Resource Locator

**XML** Extensible Markup Language

# Bibliography


[1] Soap version 1.2 part 2: Adjuncts (second edition). `http://www.w3.org/TR/soap12-part2/`, 2007. Accessed: 11. April 2015.

[2] Extensible markup language (xml) 1.0 (fifth edition). `http://www.w3.org/TR/xml/`, 2008. Accessed: 9. April 2015.

[3] Scalable vector graphics (svg) 1.1 (second edition). `http://www.w3.org/TR/SVG11/`, 2011. Accessed: 16. April 2015.

[4] Html5. `http://www.w3.org/TR/html5/`, 2014. Accessed: 11. April 2015.

[5] Hypertext transfer protocol (http/1.1): Semantics and content. `https://tools.ietf.org/html/rfc7231`, 2014. Accessed: 11. April 2015.

[6] Subbu Allamaraju. *RESTful Web Services Cookbook: Solutions for Improving Scalability and Simplicity*. Yahoo Press, 1 edition, 2010.

[7] Dana Angluin. Learning regular sets from queries and counterexamples. *Inf. Comput.*, 75(2):87–106, 1987.

[8] Günther Bengel. *Grundkurs Verteilte Systeme: Grundlagen und Praxis des Client-Server und Distributed Computing (Food Engineering Series) (German Edition)*, chapter 2. Springer Vieweg, 4 edition, 2014.

[9] E. Bozdag, A. Mesbah, and A. van Deursen. A comparison of push and pull techniques for ajax. In *Web Site Evolution, 2007. WSE 2007. 9th IEEE International Workshop on*, pages 15–22, 2007.

[10] D. Chappell and T. Jewell. *Java Web Services*, chapter 1. O'Reilly Media, 1 edition, 2002.

[11] Ecma International. *Standard ECMA-404 – The JSON Data Interchange Format*, 1 edition, 2013.

[12] Jonathan Fielding. *Beginning Responsive Web Design with HTML5 and CSS3*, chapter 1. Apress, 1 edition, 2014.

[13] Roy T. Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000.

[14] M. Isberner, F. Howar, and B. Steffen. The ttt algorithm: A redundancy-free approach to active automata learning. In *Runtime Verification*, volume 8734 of *Lecture Notes in Computer Science*, pages 307–322. Springer International Publishing, 2014.

[15] M. Isberner and B. Steffen. An abstract framework for counterexample analysis in active automata learning. In *JMLR: Workshop and Conference Proceedings. Proceedings of the 12th ICGI*, volume 34, pages 79–93, 2014.

[16] M. Kearns and U. Vazirani. *An Introduction to Computational Learning Theory*. The MIT Press, 1994.

[17] Ari Lerner. *ng-book - The Complete Book on AngularJS*. Fullstack io, 1 edition, 12 2013.

[18] Alex MacCaw. *JavaScript Web Applications*, page 98. O'Reilly Media, 1 edition, 2011.

[19] T. Margaria, O. Niese, H. Raffelt, and B. Steffen. Efficient test-based model generation for legacy reactive systems. In *High-Level Design Validation and Test Workshop, 2004. Ninth IEEE International*, pages 95–100, 2004.

[20] M. Merten, F. Howar, B. Steffen, and T. Margaria. Automata learning with on-the-fly direct hypothesis construction. In *Leveraging Applications of Formal Methods, Verification, and Validation*, Communications in Computer and Information Science, pages 248–260. Springer Berlin Heidelberg, 2012.

[21] Maik Merten. *Active automata learning for real-life applications*. PhD thesis, TU Dortmund University, 2013.

[22] M. Mikowski and J. Powell. *Single Page Web Applications: JavaScript end-to-end*, chapter 1. Manning Publications, 1 edition, 2013.

[23] N. Nurseitov, M. Paulson, R. Reynolds, and C. Izurieta. Comparison of json and xml data interchange formats: A case study. In *Scenario*, pages 157–162. CAINE, 2009.

[24] H. Raffelt and B. Steffen. Learnlib: A library for automata learning and experimentation. In *Fundamental Approaches to Software Engineering*, volume 3922 of *Lecture Notes in Computer Science*, pages 377–380. Springer Berlin Heidelberg, 2006.

[25] Alexander Schieweck. A service-oriented interface for testing web applications via automata learning. Bachelor Thesis. TU Dortmund University, 2015.

[26] Christian Schneider. Csrf and same-origin xss. `http://www.christian-schneider.net/CsrfAndSameOriginXss.html`, 2012. Accessed: 16. April 2015.

[27] M. Shahbaz and R. Groz. Inferring mealy machines. In *FM 2009: Formal Methods*, volume 5850 of *Lecture Notes in Computer Science*, pages 207–222. Springer Berlin Heidelberg, 2009.

[28] B. Steffen, F. Howar, and M. Merten. Introduction to active automata learning from a practical perspective. In *Formal Methods for Eternal Networked Software Systems*, volume 6659 of *Lecture Notes in Computer Science*, pages 256–296. Springer Berlin Heidelberg, 2011.

[29] Anne van Kesteren. Cross-origin resource sharing. `http://www.w3.org/TR/cors/`, 2014. Accessed: 11. April 2015.

[30] Stephan Windmüller. *Kontinuiertliche Qualittskontrolle von Webanwendungen auf Basis maschinengelernter Modelle*. PhD thesis, TU Dortmund University, 2014.