technische universität
dortmund

Bachelor Thesis

A Service-Oriented Interface for Testing Web Applications via Automata Learning

Alexander Schieweck
April 2015

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

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

Contents

1. Introduction

Since the early days of software development providing a good quality of the products was always a big concern. Realizing a good quality assurance, e.g. by testing the software with various approaches (e.g. unit, integration, system, user-level test) can become one of the most time consuming tasks in a development cycle. In the last years there is a movement to provide more and more applications as an online solution over the internet. Together with the growing size of software projects and the different technologies behind them, testing becomes even more complex. Providing continuous build and testing based on the user interaction with the website, is one way of taking a lot of pressure from the testing process and could still ensure a high software quality. As shown in previous papers, Active Automata Learning can be one way to provide this, but current solutions require a deeper understanding of the topic. Also the existing tools and frameworks require a high programming skill in one specific language, which must not be programming language of the main project. Most of the frameworks only provide the learning aspect (e.g. algorithm and data structures) and to simulate user interaction or other ways of interaction with an application, yet another different framework is needed, which leads to a separate technology stack. And often software is not tested by their developers, instead this task is done by a quality assurance group, which main focus is to improve and document the actual user experience and not to develop testing tools. Therefore these groups do not have the experienced developers needed and wont get them. This means that as of today no solution for Active Automata Learning as a testing component can be practically used by a bigger target group.

That this problem actually exist could been seen in the summer 2014 iteration of the lecture *Webtechnolgien II* (engl.: *Web Technologies II*), an advanced lecture in the second half of the *Bachelor in Computer Science* program at *TU Dortmund University*. The content of this lecture is to teach the basics of a enterprise like web application with Java. A topic that cannot only be taught trough slides, so the students were asked to come together in small groups and to develop a simple application. The task was to write a small ToDo-Application, which allows registered users to create, view, edit and delete tasks. To test the application the students should not only use common tools but also use Active Automate Learning. Only a few had used Java Enterprise Edition (Java EE) before and for most them the introduced frameworks and libraries challenged them on their own. In the end even the skilled computer science students, which overall liked the

idea and tasks, had trouble with the learning aspect and it was only implemented by a few groups.

Alexander Bainczyk and I attended the lecture back then. We managed a good solution of the given tasks (see figure 1.1) and after the semester a few members of the Chair for Programming Systems asked us, how the lecture could further improve, especially how more students could fulfill the learning task. This was the point when the idea was born to create something more simple, which would allow the use of Active Automata Learning for a bigger target group. Something that could be used in the next iteration of *Webtechnolgien II* and in a real world application, maybe even to test large, enterprise business software. The larger goal is to simplify active automata learning for testing web applications, which means it should be easy to use, so that the entrance barrier into this field is as low as possible. On the other hand it should allow powerful features and hiding too much to the user, so that it is also useful to experienced users. Also this could make the tool even more interesting for new users, who like to explore and experiment.

To achieve this idea Alexander Bainczyk and I decided to create ALEX and we decided to go with a web based application again. This allowed to split the project into two sub-projects: The back-end and the front-end part. His main focus is to create a modern browser based front-end and it is recommended to take a look at his thesis [2], too. My thesis will cover the back-end, which will be realized as service-oriented interface.

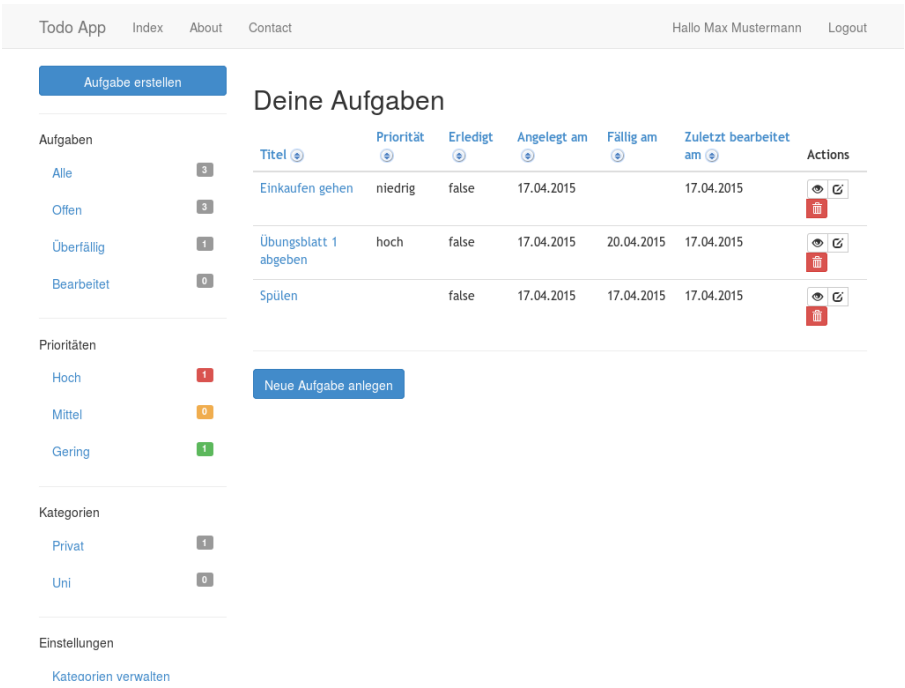

Figure 1.1.: The Main View of our Todo-App.

1.1. Goals of this Thesis

A service-oriented interface has some advantages besides the possibility to split up the project. Other aspects which can be realized through such an architecture are:

Language independent ALEX should be an abstraction layer between the user and the Active Automata Learning. This should also allow person without huge developing skill to create and set up a learning process.

Persistence It should be possible to persist the set up and the learning results. This is necessary because it is a web application. This creates the opportunity to compare different set ups and to see how a project evolved over time.

Standalone ALEX should make things easy, not complicated. So it should be easy to run it without a huge installation process. Providing it as a service-oriented interface reduces the set up to one server and reduce the installation overhead. This implies also that the service-oriented back-end can run without any front-end (see figure 1.2).

Well documented Providing a good documentation can make the difference between a bad or a good software. This is especially true for service-oriented interface, where it is not clear, who will use it to create some sort of front-end.

Extendable There are many different ways to interact with an application and in this thesis only a few will be considered. So it should be possible to extend ALEX without too much effort. This should also apply for Active Automata Learning algorithms.

1.2. Thesis Structure

Following this motivation, the upcoming chapter will give some insights about some basic concepts and frameworks. This includes a short look on the big topic of (active) automata learning in section 2.1 and an introduction into service oriented interfaces in section 2.2.

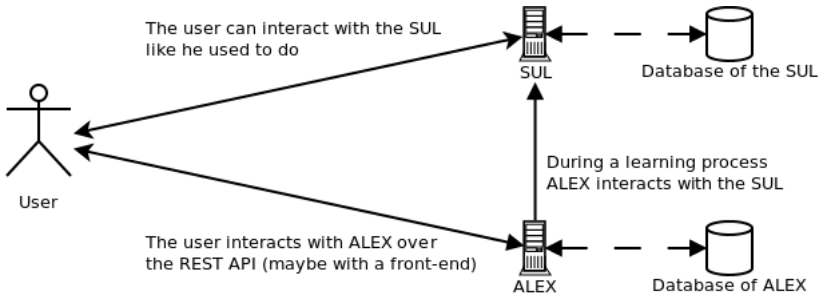

Figure 1.2.: General Architecture of ALEX.

Additionally some more tools and frameworks will be presented in section 2.3. After this in chapter 3 the knowledge from the basics chapter will be used to design a service-oriented interface around a learning process for web applications. Following this design process, some of the main aspects of the implementation process will be shown in chapter 4. In chapter 5 the ALEX will be used to test the ToDo-Application and to compare the new approach with the old one. At the end in chapter 6 it will be time to look back at the project and close of with some ideas on improvements.

2. Basics

This chapter deals with the different pre requirements, techniques and researches used and referenced in the rest of this thesis. This includes a introduction to automata learning and a look at service oriented interfaces. At the end some frameworks and tools will be presented, that will be useful in the implementation phase.

2.1. Learning for Testing

First lets take a look at the general topic of automata learning with some of the common approaches and algorithms, including the passive and active form. Afterwards it will be discussed how this could be used for testing purposes and will then be compared with some common testing techniques with the focus on web applications.

2.1.1. Introduction into Automata Learning

Automata Learning can be used to get an understanding of a system, which is not well known, i.e. a system we have no information about its insight and cannot take a look at it. So observing the input and output behaviour of such a "black box" system is often the only way to get information about its inner structure. This information can be used to create a model of the System under Learning (SUL) and in an ideal world the model will fit perfectly. This model could then be used to validate the running SUL instance against its specifications and requirements.
To represent such a model a Mealy Machine is often used, because it is a small and well defined data structure, which allows e.g. model checking to verify the behaviour.

2.1.1 Definition. A Mealy Machine $M = (Q, q_0, \Sigma, \Lambda, \delta, \lambda)$ where

- Q is set of states.
- q_0 represents the starting state.
- Σ is an input alphabet (set of input symbols).
- Λ is an output alphabet (set of outputs symbols).
- $\delta : S \times \Sigma \to Q$ is the transition function.

- $\lambda : S \times \Sigma \to \Lambda$ is the output function.

The meaning implied by such model of a SUL is that every state represents a state of the system. Each transaction is triggered by user interaction $\alpha \in \Sigma$ and the system reacts to this input and will answer with corresponding output $o \in \Lambda$ defined through the output function λ. This can be combined with a state change according to the transition function δ. For an example please refer to figure 3.1.

Passive Learning

One possible method to get information over the input and output behaviour is to let the program run in a production like environment and to write every user action and system reaction into a protocol. Many programs have already some sort of logging implemented, so this is not to hard to do. The intention behind the logging is to find errors or problems in the production environment and to allow a faster reaction to them. This log files can also be used to analyse user behaviour or to guess the target group of a web application. This leads to the problem that those log files often tend to grow very big, so selecting the relevant entries for the model creation is hard. And even if the files are big, they often only contain data about usual use cases and some possible ways of interaction are never tested. [14].

Active Learning

Another approach is to create inputs for the SUL and to simulate user activity. This way not only usual and intended interaction with the application are observed, but also rare and even meaningless use cases. One way to implement this is using the minimally adequate teachers (MAT) introduced by Dana Angluin [1]. This model has three parts: The “teacher” as main component which uses a membership oracle and an equivalence oracle. The membership oracle is the part between the teacher and the SUL. It maps the given input to some action on the SUL and answers with the corresponding output. The equivalence query is then used to check the created hypothesis against the SUL and to answer with an counterexample to prove the difference between the models. If not already a perfect model of the SUL is available, the task is semidecidable, which means that if there is a couterexample it is possible to can find one and the algorithm will terminate eventually, given possible infinite amount of time. If the model is perfect and no counterexample exists, the algorithm will run infinitely. To be usable in practice, the algorithm will be forced to stop at one point. If a counterexample was found up to this point, the model is not exact and can be refined. On the other hand if no counterexample was found this could mean that the model is perfect or that the search algorithm was stopped to early and had just not enough time to find the counterexample. So the equivalence oracle can only be approximated by using the membership oracle, too.

The learning can then be done in a learning loop shown in algorithm 2.1.

```
hyp = learner.startLearning();
repeat
  hyp = learner.getHypothesis();
  ce = eqSearch.findCounterexample(hyp, alphabet);
  if ce != null then
    learner.refineHypothesis(ce)
  end if
until ce != null
```

Algorithmus 2.1: A Simple Learn Loop

The first step in this loop is to learn a first, initial hypothesis (line 2). After this a counter example i searched (line 5). If one counterexample is found, the hypothesis will be refinded to respect the new counterexample. This loop is repeated until no counterexample is found. Again, this does not mean that there is none, it only means that the equivalence oracle did not find one within the given parameters for the it.
To have an initial state it must be possible to reset the system and how this reset is done depends on the SUL. For example if the application uses a database this could be done by clearing the whole database, which can take some time. Other strategies could be to just ensure that each iteration is independent from previous ones. For example if all action will only affect one user at a time, creating a new user in each iteration could be a faster way to go.
One of the first algorithms was the L* as shown by Dana Angluin in 1987 [1]. This algorithm was only for deterministic final state automata but could be extended to use mealy machines [12]. This algorithm is based on an observation table, which contains two word sets S and $S\Sigma$ and a suffix set E. For every word se with $s \in S \cup S\Sigma$ and $e \in E$ the last output will be saved in the corresponding cell. After the table is filled a hypotheses can be created out of it. An example observation table is presented in table 2.1.
Many algorithms like the DiscriminationTree [10] or DHC [21] followed and the newest one is the so called TTT algorithm, which is very space efficient. [9]. All those algorithms have different characteristics in which they differ from each other. This characteristics include

		E		
		A	B	C
S	ϵ	OK	OK	FAILED
	A	OK	FAILED	OK
	B	FAILED	OK	OK
$S\Sigma$	C	FAILED	FAILED	OK
	AB	OK	FAILED	FAILED
	BB	FAILED	OK	OK

Table 2.1.: Example observation table

statistics like the the amount of membership queries or equivalence queries needed, the duration of the learning process or the memory usage.
Some of them can be improved by a cache or a reuse filter [3]. Cache just saves the query send to a membership oracle and the corresponding answer. If the exact query is requested again, the cache can answer it without interaction to the SUL. This increases the performance because the SUL is mostly slower than the learner. Taking this idea a step further, a cache that would not only answer complete queries but also could reuse existing system states can provide further performance improvements. If a hard reset is done, e.g. clearing a database, this task gets more complicated. Running multiple instances of the SUL to keep different states alive over a reset, e.g. by starting a new instance one each reset, can help in that case.

2.1.2. Tests of Web Applications

First lets take a look into what a web application actually is. For the purpose of this thesis a web application is a piece of software that runs on some kind of server and can be used through a web browser. The software will react to input given by users and it can handle multiple users at once. Furthermore the behaviour should be consistent and deterministic. The last two properties also apply to a bigger group of systems, called reactive systems. To define a web application it is not relevant if the server runs in a data center or is just a local one. Most of these web application are based around common protocols, like HTTP, and often offer more than one interface to interact with them. An example of a typical web application is the ToDo-App introduced in chapter 1.
Testing is an essential part of software development and often takes a huge part in software projects. While it is obvious that an application should be created, which has no bugs at all, it is hard to achieve. Web applications are no exception to this general rules. Taking a look at often used techniques to ensure a good software quality:
The most basic one is the *Manual Testing*: A developer checks a feature in the software per hand. This is fairly easy and the programmer gets a feeling of his project. The downside is that it take a lot of time. If the project get bigger it is hard to check ever possible input and check the reaction of the system. It takes at least a lot of time and therefore the testing is not done very often which leads to bad quality. Additionally the developer is not objective and could be tempted to avoid broken parts of the program.
One popular option are unit tests. These are test where only a small piece, e.g. a class in object-oriented programming, is tested against the specification. Often it is impossible to test against all possible cases, e.g. to test every integer value, most of the times testing special edge cases, e.g. 0 and "MAX_VALUE", gives enough confidence that the tested part works as intended. Those test run fully automatic and can be executed often. This increases the chance to find conflicts of new implementations and changes, that could affect

other parts of the software. But even if all test are running OK there is still the possibility that two pieces will not work together the way they are supposed to do.
This leads to *Integration test* where the complete software is tested. Often this is done by trying to execute typical use cases, which would be done through manual testing, but now in an automatic fashion. This means that the test are more likely to find bugs the user would encounter, too. This automation helps to reduce the work load of the tester and reduce the likelihood of forgetting or excluding a part from the testing process, which might be buggy.
Both of the described techniques have advantages and disadvantages, but they work good together. Integration test can not be used efficiently, if the program is not working at all. But only relaying on unit test might lead to an unstable product as well. However there is still one bigger problem with this mix: The integration tests only test specified use cases. While this is an improvement to the complete manual testing, it would be better to test as many use cases as possible to ensure that even unexpected behaviour will be handled accordingly.
This is possible with Active Automat e Learning as a new way to implement integration test. The typical actions a user could do, e.g. register or login, can be used as an input alphabet and the learning algorithm will execute them in any possible way until a hypothesis of the application is created. This mealy machine can than be used to validate the SUL. This can be done by checking special constrains or by comparing the new result to previous results [22].

2.1.3. Previous work

In the past some applications were and still are tested by Active Automata Learning. For example the Online Conference System (OCS) is a enterprise level web application developed at the Chair of Programming Systems and often used an example for this. Additionally Stephan Windmüller presented in his PhD thesis [22] another web application: He tested the bug tracker *Mantis*. Furthermore he looked closer on how to test the different interfaces of a a web application and how to compare them. Also he looked on how to use Active Automata Learning to compare different versions of the software and the learned models.
To make those experiments easier to set up a framework called *LearnLib* was develop [21, 15]. This framework provides for example an collection of algorithms or a bunch of equivalence oracles. A few years back only a closed source version existed, but in 2013 a complete reimplementation as an Open Source project[1] was released. This new *LearnLib* is the core of the learning part of ALEX.

[1]Licensed under the GNU Lesser General Public License (LGPL)

2.2. What is a Service-Oriented Interface?

The idea of Service-Oriented Interfaces is not new and it can be considered as part of a service-oriented architecture. This section gives a overview about two popular standards, SOAP and REST, and at the end decision will be presented, which of these fits better for this project.

2.2.1. Advantages of a Service-Oriented Interface

A service-oriented interface is often based on a server client structure to make front- and back-end independent from each other. This allows to scale the application to a much higher level and releases the restriction to just to just one front end. The variety of modern front-ends is big, including classic desktop applications, web applications or apps for Smartphones. Supporting all these different concepts, is most likely not possible for one developer team. A service-oriented interface can be used by other developers to write a front-end for their needs and even include the service in their products. Often a simple and common way to interact with a service is one requirement, so that no special tool kit is needed in order to use the service-oriented interface, This means that they are often build on existing technologies, like Hypertext Transfer Protocol (HTTP).

2.2.2. SOAP

SAOP is a communication protocol specified and is a successor of "XML-RPC"[2]. Version 1.0 was released in 1998 by Dave Winter and some engineers of Microsoft. After this other companies like IBM started to support the idea and it became a W3C standard. Originally it SOAP was an abbreviation for "Simple Object Access Protocol" but this phrase did not get the point of it. So SOAP has become its own trademark with its own meaning. It defines its own protocol on top of other application layer protocols like HTTP, with strict definitions of terms and it uses XML to transfer data.

2.2.3. Representational State Transfer

REST was defined by Roy Thomas Fielding in his PhD thesis in 2000 [5]. It is more flexible protocol than SAOP and is based on the HTTP protocol. To outline REST Fielding defined six core principles:

Client-Server The REST idea describes a simple client and server architecture.

Stateless The communication is not based on states or sessions. This allows to scale the system, because a new server can be added whenever needed without running into session synchronization troubles.

[2] "Remote Procedure Call" over Extensible Markup Language (XML)

Cache It should be possible to use a cache (on client or server side) to improve the performance.

Uniform Interface REST should be done via a clear interface.

Layered System To allow complex system and higher performance. The client only talks to one layer on the server side. The other layers are hidden to the outside.

Code-On-Demand Request to the server to execute some code. This is the only optional participle.

Every REST offers different URLs to interact with the service. Often those URLs are not static, but contain parameters. When talking about such URLs the parameters are often highlighted with curling braces, e.g. the URL "/project/pid" contains a parameter with the name "pid". To describe what action should be performed the HTTP methods GET, POST, PUT, DELETE are used:

GET Receive information, changes nothing.

POST Asks the server to create a new entry or to do some action.

PUT Update existing data or store data in a store, i.e. without checks. Sometimes this is also used to create a large amount of data objects.

DELETE Remove some object from the database. The data will be actually removed, so now undo possible. This operation is idempotent, i.e. calling DELETE once, twice or n-th times has the same effect: The data is gone. A new request may return another status code, like 404, after the first request, but the stored data is not changed.

The HTTP response contains status codes, which are used to make the outcome of a request clear (see table 2.2) [13].
REST is not a strict protocol but more a collection of general rules and best practices approaches. It only requires to use a standard message format, like XML or JSON, but it is possible to choose any format that fits the problem. Currently there is a trend to use prefer

Status Code	Status Message	Recommended usage in REST
200	OK	General OK/ success message.
201	CREATED	Often send when something was created.
204	NO CONTENT	No body, often send when something was deleted.
400	BAD REQUEST	The request had the wrong structure.
403	FORBIDDEN	The client hast not the right security level
404	NOT FOUND	The requested resource was not found.

Table 2.2.: Most common HTTP status codes [19]

JSON overXML. This trend is based on the expanding market for SingelSiteWepappplications which. JavaScript offers native JSON support, while XML must be parsed. A typical JSON document follows the JavaScript notation of an object, which is always introduced with a pair of curling braces "{ ...}". Inside this different properties can be specified and each property has a name given in quotation marks and a value, separated by a colon. Different properties are then separated by a comma. Often additional whitespace is inserted to improve the readability, but this is not necessary. The value of the property has to be one of the following types: Number, Boolean, String, Object, Array or Null. A Number could be an integer or a floating point value. As Boolean property is only true or false. A String is surrounded with quotation marks like the property name. An Array start with "[" and ends with "]" and can hold every data type. The elements within an Array should have the same type and they do not have a property name (listing 2.1) [20].

```
{
  "name": "Alice",
  "age": 42,
  "friends": [1, 2,, 3]
}
```

Listing 2.1: JSON example.

Why ALEX uses REST

Based on the characteristics of SOAP and REST and the project goals, it seems that REST is the better choice for the project. It is more flexible and faster to parse [16]. Additionally this project should be easy to integrate in existing set ups and For this case a simple HTTP call to a RESTful service is a better choice, because a small HTTP request is simpler than working with a SOAP layer on top if. This means REST will be used for this project and from now on only REST will be considered in this thesis.

2.3. Frameworks

To implement or support the previous introduced technologies a lot of tools, library's and frameworks can come in handy. The more important ones are listed in this section and this list is by far not complete. The creation of a full list would not provide any more necessary information. For further information please consider to take a look at the project itself.

2.3.1. Maven

Maven[3] is build tool for Java and takes care of dependency management too. This means *Maven* can download all dependencies of a project. Compile the source code and run the test. If everything is OK it then can package all together in a *.jar* or *.war* file.

To allow even bigger projects *Maven* supports so called multi-module projects. This means that a project can be slitted into different "*modules*". Those modules help to keep the project structure clean and makes it easier to reuse some parts because they can be included as a module like every other one. *Maven* also supports inheritance so common properties or settings can be bundled in a parent module.

Furthermore it is relatively easy to create a project documentation with *Maven*. This feature creates a website which can contain custom pages as well as automated generated reports. It is possible to write an own site in different formats like Markdown which than will be compiled into static HTML. The automated reports are generated by different plugins and can contain information about the used dependencies, the source code documentation from JavaDoc, an overview about the code style qualities calculated with *Checkstyle*[4] or to do some static code analysis with *Findbugs*[5]. Also there are multiple, different styles or "skins" to choose from to create a custom look.

2.3.2. Jetty

Jetty[6] is a lightweight web server written in Java. It can be easily integrated into other projects. Furthermore there is a good *Maven* plugin available, which allows to run jetty directly from the build tool. Running the application and setting up the developer environment is done with just one command. Additional *Maven* can be configured to start Jetty for integration test. The down side of Jetty is, that is it not a full Java EE application sever, so that using other technologies are needed.

[3] `https://maven.apache.org`
[4] `http://checkstyle.sourceforge.net`
[5] `http://findbugs.sourceforge.net`
[6] `http://www.eclipse.org/jetty`

2.3.3. Hibernate

Given some Plain Old Java Objects (POJOs) that should be persist in a relation database, the obvious way is to write all the SQL statements and queries. This can get out of hands. Hibernate[7] can solve this problem, because it is a so called Object Relation Mapper (ORM) Mapper which can map the POJOs to a relational database schema. So the inserting, selecting, updating and deleting can be done with some method calls to the Hibernate API and Hibernate implements the Java Persistence API (JPA). Hibernate supports a variety of different relation databases, including *HSQLDB*[8], which can run as memory only database. Because Hibernate does all the SQL there is no need to worry about the slightly differences in the SQL syntax introduced by every database system [11].

2.3.4. Jersey

Jersey[9] was a reference implementation of the Java web service API. It offers a full JAX-RS implementation flowing the standard rules defined for Java in JSR 311 and JSR 339. Also Jersey offers a nice testing framework to test REST resources in unit test without setting up a septate server.

2.3.5. Jackson

Jackson[10] is used to serialize and deserialize Java objects to and from JSON. Jackson provides a huge API of parsers and generators around the JSON format. Similar to Hibernate Jackson provides an easy way to use annotation to customize the (de-) serializing process into or form JSON and to select, rename or ignore some properties. Jersey and Jackson work great together and most of the time it is not needed to talk to the Jackson directly because Jersey will take care of it.

2.3.6. Selenium

Selenium[11] is a framework to simulate how a user interacts with a web application through a browser. To simulate this *Selenium* starts a browser and sends commands of a *Webdriver* to this instance which then will be executed by the browser. The set off supported browsers include the popular *Mozilla Firefox*[12] or *Chrome*[13]. But it is very unlikely that such a desktop browser could run on a server, because they are mostly text based. For this case

[7] `http://hibernate.org`
[8] `http://hsqldb.org`
[9] `https://jersey.java.net`
[10] `https://github.com/FasterXML/jackson`
[11] `http://www.seleniumhq.org`
[12] `https://www.mozilla.org/de/firefox`
[13] `https://www.google.de/chrome`

the “HtmlUnitDriver” is an possible way, because this driver starts a headless browser without any GUI.

3. Designing an API for Active Automata Learning

As mentioned in section 2.2 a REST API is tool for other developers. So it is beneficial to design the API first and then implement it. Following this principle this chapters describes how a RESTful service should look like to provide a platform for automata learning.

3.1. Learning a Web Application

The main focus when talking about learning a web application is how to interact whit it and how this interaction can be defined through JSON. The other components like algorithms are provided in the LearnLib and only require a few parameters, which can be simple properties of a start command. Symbols will then not only be used a usual input symbols to the learn algorithm, but also to reset the SUL.

From a technical perspective it looks obvious to strictly separate test cases of the web and REST interface, because different techniques are required to interact with them. Most of the previous works follow this approach, which leads to the problem that it is not easily to compare the test results from the different ways to interact with a web application. Also from the perspective of the web application it should not matter how the interaction with it happens. So this is not a good way to simulate different user interactions.

This leads to the idea of allowing Web and REST symbols together in one learning process. Web symbols in this case are symbols, that contains different action to interact with the application over the web interface. REST symbols are the equivalent for the REST interface. If both types are present in one learning set up, a way to exchange data between them is needed. Also it is possible define one use case of a web application in two different ways and compare the behaviour of the web and REST interface in one test run.

To talk easier about symbols, which are doing the same changes to a system, but maybe over different layer, the term of *impact equivalence* is fitting. Symbols are called impact equivalent if and only if they have the same impact on a bug free system and their output behaviour is identical. This means if a hypothesis $H^* = (Q^*, q_0^*, \Sigma, \Lambda, \delta^*, \lambda^*)$ of a bug free system, which is generated by using the symbols Σ, is given, then $\forall x, y \in \Sigma : \forall q \in Q^* : \delta^*(q, x) = \delta^*(q, y) \wedge \lambda^*(q, x) = \lambda^*(q, y) \Leftrightarrow x \simeq y$.

For example figure 3.1 shows a model of a bug free system. To create this model four symbols were used $\Sigma = \{input1, input2, input3, input4\}$. The input symbols "input1" and "input2" are equivalent, because they have the same transitions between the states and have the same output behaviour. "input3" is not equivalent to them, even if the transitions are equivalent, because the output is different. And "input4" describes a complete different be behaviour.

This definition leads to different observations. If Σ is given and an additional symbol $x \notin \Sigma$ exists, so that $\exists y \in \Sigma : x \simeq y$, y can be removed from Σ and instead x can be used. The resulting hypotheses will be isomorph to the original and the bijective function behind the isomorphism is essentially just to replace y with x. Taking this a step further, all symbols in Σ_w can be replaced by Σ_r, which only holds symbols that are equivalent to those in Σ_w. This is just a different way to talk about the idea, that the models of web symbols and REST symbols should be isomorph. If we now use $\Sigma_w \cup \Sigma_r$, we will end up with an model, where every transition is "doubled" and both models are laying on top of each other. Thinking the other way around this allows to check quickly if the symbols from Σ_w and $Sigma_r$ are equivalent without setting up two learning processes and without complex graph algorithms.

This mix of interfaces allows a more realistic communication with a web application, because using the different ways strictly separated, is not the idea why a system offers them and it should not be a problem to use both at the same time. However it is more complicated to reset the SUL, because a reset must maybe done via the web interface and via the REST interface, which leads to at least two reset symbols. In this case some order must be introduced to determine which reset is executed first. Otherwise it could may would reset the database twice and lose reset data created over the other interface. This implies also that one reset symbol maybe needs data from the other reset symbol, e.g. if a new user is created over the REST API and must be confirmed with a code over the web interface. In the end a large amount of reset symbol would be needed if an alternation between the different interfaces is required for a successful reset. This is a lot of overhead and is not very practical.

The previous approach can be generalized even more if it is allowed to use different action types within one symbol. This is an extension to the last ideas, because it is still possible to only use one type per symbol. A way to exchange data between the different types is

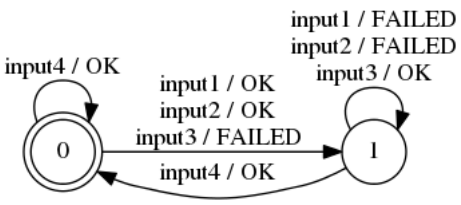

Figure 3.1.: Example of a Mealy Machine with two equivalent & two equivalent symbols..

still needed and the main benefit here is that only one reset will be needed, because the alternating list can be executed directly in it.

3.2. Data Collections

To handle all the necessary data to start a learning process and to view the resulting hypotheses a variety of data collections are needed, which can be created, read and modified through different operations. Those data collection are projects, symbols, symbol actions and symbol groups. Projects are the top level structure and contain all the other data structures. Symbols provide the main data for a learning process and every symbol contains different actions. Symbol groups are also part of a project and contain symbols, but their main purpose is to allow the user to keep the symbols organized.

3.2.1. Projects

A project is the object of our data collections and it contains all other data structures, like Groups, Symbols and Learn Results. The primary goal of a project is to organize different test set-ups within ALEX and to store central data, like the base url of the application to test. To keep it simple to differentiate projects the name must be unique. A sample project definition is shown in listing 3.1.
The operation needed for projects are the typical Create, Read, Update and Delete (CRUD) operations (see table 3.1). Deleting a project also removes all data belonging to it.

3.2.2. Symbol Groups

When setting up a learn process for a larger application it is likely to create large amount of Symbols. To keep the Symbols organized Symbol Groups are introduced, which allow to group symbols together and to name this group. A sample group is presented in listing 3.2.
Moving one symbol from one group into another implies that all revisions of the symbol are moved. Deleting a group will move all symbols of it into the default group of the Project. It should not be possible to delete the default group. And lastly there should be way to

HTTP Method	Path	Description
POST	/projects	Create the project in the request body.
GET	/projects	Get all projects
GET	/projects/{pid}	Get the project with the Identifier (ID) “pid”
PUT	/projects/{pid}	Update the project with the ID “pid”
DELETE	/projects/{pid}	Remove the project with the the ID “pid”

Table 3.1.: HTTP resources for Projects.

```
{
  "id": 42,
  "name": "The Project",
  "baseUrl": "http://localhost",
  "description": "An example project."
}
```

Listing 3.1: Sample JSON of a Project.

```
{
  "id": 42,
  "name": "The Project",
  "baseUrl": "http://localhost",
  "description": "An example project."
}
```

Listing 3.2: Sample JSON of a Symbol Group.

fetch all symbols of a group. For this the path "/projects/{pid}/groups/{gid}/symbols" is chosen. Table 3.2 gives an overview over the API endpoints.

3.2.3. Symbols

Symbols are the core of the interface and the main part of every project. It is identified by an ID within the project and its name and abbreviation must also be unique per project. The difference between the name and the abbreviation is, that the abbreviation is restricted to 15 characters and will be used in the models, which are more readable because of this restriction (see listing 3.3).
The interface should allow to create and update a symbols and to fetch a list of symbols or just one *Symbol*. When asking for a list of symbols it should be possible to use some filter. *Symbols* should not be removed from the database, because it could be used for a learning process or referred within an Action, thus it is now possible to just modify or delete a *Symbol*. Therefore updating a symbol creates a new revision of a *Symbol*, so that the old version is still there. Instead of removing it from the database we use a hide and show concept (see table 3.3). Because often a large amount of *Symbols* should be managed, an API for batch creation and batch update is also provided.

HTTP Method	Path	Description
POST	/groups	Create the Symbol Group in the request body.
GET	/groups	Get all Symbols Groups in the project with the ID "pid"
GET	/groups/{gid}	Get the project with the ID "gid"
PUT	/groups/{gid}	Update the Symbol Group with the ID "gid"
DELETE	/groups/{gid}	Remove the Symbol Group with the the ID "gid"

Table 3.2.: HTTP resources for Symbol Groups. All paths have the prefix "/projects/{pid}".

```
{
  "id": 21, // The ID within the project
  "project": 42 // The ID of the project the symbol is part of.
  "group": 1 // The ID of the group the symbol is part of.
  "name": "A Symbol", // The name of the symbol..
  "abbreviation": "s1", // The abbreviation used in the models
  "actions" : { // actions of the symbol. Please see below.
    ...
  }
}
```

Listing 3.3: Sample JSON of a Symbol.

HTTP Method	Path	Description
POST	/symbols	Create the symbol group in the request body.
GET	/symbols	Get all symbols in the project with the ID "pid"
GET	/symbols/{sid}	Get the Symbol with the ID "sid"
PUT	/symbols/{sid}	Update the Symbol with the ID "sid"
DELETE	/symbols/{sid}	Remove the Symbol with the the ID "sid"

Table 3.3.: HTTP resources for Symbols. All paths have the prefix "/projects/{pid}".

3.2.4. Actions

Actions are extruded sequentially in every symbol. If an action fails, the outcome of a symbol is already clear and all action after the failing on will not be executed. This behaviour can be changed by providing a "ignore failure" flag in which case an error will not set the symbol outcome to "FAILED" and all following actions will be called (see figure 3.2).

Also there should be a option to negate the result of an action. Every action is identified by the *Project*, *Symbol* and its number within the *Symbol*.

Like often mentioned before there should be three types main types of actions:

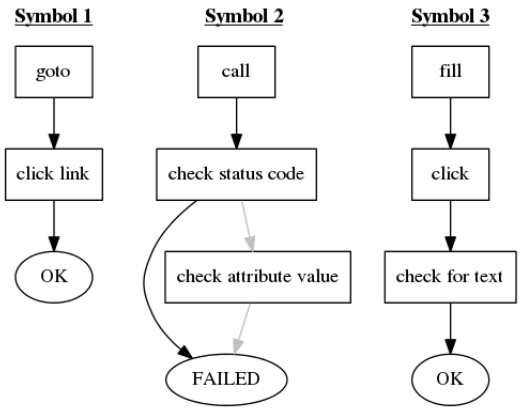

Figure 3.2.: The word "symbol1, symbol2, symbol3" with the output "OK, FAILED, OK"

Web These groups of actions interact with the web application through a browser.

REST Interact with a REST. Currently only responses in JSON format are supported.

Miscellaneous Collection of various actions. Most of them allow to interact with a variable storage.

Counters are used with "#counter" and variables have the syntax "$variable". This will be replaced in strings. No indirect addressing, i.e. counter in variable names. Also variables are only per learning process, while counters are persisted in the database. When resetting the SUL is done by just incrementing counters, a hard reset would be needed after each learning process. Keeping the counters persisted over all learn results allows to let the SUL running while doing a sequence of different test runs.
For an complete overview about the actions please take a look at table A.1. This set holds a mix of different aspects and allows to test most features of a web application.

3.2.5. Learning Results

After learning the created model will be stored in the database and should be accessible for the uses. To provide a finer resolution of this data they should be saved per iteration of the learn loop (algorithm 2.1), which will be called a "step". The step with the number 0 will hold a sum of all statistics, so it is easy to fetch an overlook over the complete learning process.The learner will create and update the results in the database, so it will not be possible to create or update such a result over the interface. Only some fetch operations should be implemented and the option to remove a complete test run. Removing only a single step would also be a manipulation of the results and should therefore not happen (see table 3.4).
Beside this hypothesis the configuration, which was used to start the learning, should be stored, so that is easy to keep track of different test set ups. Also some interesting statistics should be provided, like the amount of membership queries, the amount of equivalence queries, the overall duration of the learning process and how many individual symbols were called (see listing 3.4).

HTTP Method	Path	Description
GET	/results	Get all final learn results of the Project.
GET	/results/{testno}	Get one final learn result.
GET	/results/{testno}/complete	Get all steps of a learn result.
DELETE	/results/{testno}	Remove all steps of a learn result.

Table 3.4.: HTTP resources for Learning Results. All URLs have the prefix "/projects/{project_id}"

```
{
  "id": 21, // The ID within the project
  "project": 42 // The ID of the project the symbol is part of.
  "testNo": 1 // The number of the test run.
  "stepNo": 1 // The number of the step within the test run.
  "hypothesis": {
    "nodes": [0, 1] // all nodes of the model
    "edges": [ // the transition between the nodes
      {"from": 0, "to": 0, "input": "s1", "output": "OK"}
    ]
  },
  "statistics" : { // some statistics
    "#mqs": 10,
    "#eqs": 10,
    "startDate": "2015-04-17T00:00Z", // ISO 8601 conform
    "duration": 12345 // in milliseconds
  }
}
```

Listing 3.4: Sample JSON of a Learning Result.

3.3. Learning Control

Finally commands to control a learning process are needed. Those commands need to know which symbols should be used during the learning, which symbol does the reset, which algorithm to use and which strategy to find counter examples should be used. Additionally the amount of steps to learn with the given settings can be restricted, if learner should not repeat until the equivalence query finds no more counterexample. First we have a command to start a learn process. Providing the symbols to use, the reset symbol and the way a counterexample will be searched a learning thread will be created and the learner is active. Then there should be a command to resume a learning process. This can be done with some different configurations, but it is not possible to change the algorithm or symbols, because the resuming is during a learning process. The last command is to stop the learning after the current step. Allowing a direct abort could leave the algorithm in an unclear state, which would not allow the resuming command. It should be possible to get the output of a manual counterexample by providing just the symbols and the learning will call them in order to determine the output in an automatic fashion (see table 3.5).

HTTP Method	Path	Description
POST	/learning/start	Start a learning process.
POST	/learning/stop	Stop the learning after the current step.
POST	/learning/resume	Resume the last learning process.
GET	/learning/active	Is a learning process active?
GET	/learning/status	Get the current Learning Result as status.

Table 3.5.: HTTP resources for Learning control.

3.4. Error handling

While researching the error handling process of a service-oriented interface it seems that there is no best practice way. A common opinion exists that an error should be handled with a clear error messages, but everyone differs on how such a message looks like [18]. To come to a format for the error handling of ALEX, some research and inspiration on large REST interfaces is done:

Facebook Facebook uses a larger JSON response with different internal data and status codes. [4].

Twitter Uses different HTTP status codes and a human readable error massage: '{“errors”:[{“message”:“Sorry, that page does not exist”,“code”:34}]}'. Note the internal error code which can give even more information about the error, if the error codes in the API documentation are looked up[8].

Twitch Uses different HTTP staus code ad a human readable message: '{“message”:“Invalid Token”, “status”:401, “error”:“Unauthorized”}'. The message is not wrapped into a field called “error” and the provided status code matches with the one in the HTTP header. An usual error code with an good description helps to understand the API without looking into a documentation [7].

The idea of presenting the HTTP status code in the response body seems helpful, because if the user is not watching the HTTP response header, this information could be lost. Defining own error codes on top of this is a bit to much. Providing a good error message should lead to the cause of the error without reading a documentation to understand some arbitrary number. So in the end the error format looks like the example in listing 3.5.

```
{
  "statusCode":  404, // The HTTP status code
  "statusText": "NOT FOUND", // the corresponding text
  "message": "The recourse with the ID 42 was not found."
                                    // a descriptive error message
}
```

Listing 3.5: Sample JSON of an error.

4. Implementing the API in Java

After defining the service-oriented interface in the previous chapter it is time to implement it. This is done by using the frameworks introduced in section 2.3. This chapter is limited on the key features and interesting side projects. So please fell free to check out the code by yourself for further details.

4.1. Java Objects, the Database and JSON

One of the key features is to persists the data in a relational database and to read and send them encoded in JSON.

To have access to the data in the code a simple Java class is required, which only holds the data and contains no business logic at all. Such a class is often called an *Entity* (see listing 4.1). To provide further details about the Entity and their members, annotations are used. Those annotations can be read during the compile or run time by frameworks, which then handle the *Entity* accordingly.

```
1  @Entity
2  public class Symbol {
3      @Id
4      @GeneratedValue(strategy = GenerationType.TABLE)
5      @JsonIgnore
6      private Long id;
7
8      @ManyToOne
9      private Project project
10
11     @NotBlank
12     privet String name;
13
14     /* Insert Getter & Setter here */
15 }
```

Listing 4.1: A Sample Entity Class.

4.1.1. Realizing Persistence

The default way to interact with a relational database is to write statements in the database language SQL. However this method has some major disadvantages: Firstly another language is introduced in the project and a large amount of queries has to be written. A change at the *Entity* code could then require a lot of changes in those SQL queries. Secondly, most popular database systems have some special, non standard conform interpretation of SQL, which would require additional code to support multiple database systems or to limit the project to just one certain database system. *Hibernate* can help to solve those issues. IT is possible to configure *Hibernate* to use various database systems. Provided a list of Entities to store, *Hibernate* checks theses *Entities* for special annotations and creates the database schema for it. Firstly every *Entity* has to have the "@Entity" (line 1) annotation and an unique ID ("@Id", line 3). This ID can be auto generated by the database ("@GeneratedValue", line 4). *Hibernate* also supports relation ships between entities (e.g. "@ManyToOne", line 8). It is also possible to exclude fields from the persistence using the "@Transient" annotation. Furthermore *Hibernate* can provide basic validation of the properties, like forbidding null values ("@NotNull") or enforce non blank strings ("@NotBlank", line 11).
To manage the *Entities Hibernate* provides a special API. To encapsulate this API at one place a layer of Data Access Objects (DAOs) is introduced (see listing 4.2). Those take care of creating, fetching, updating and deleting *Entities*. This layer also validates the data and adds the IDs a *Symbols* or *SymbolGroup* has within the project. Besides the default validation provided through *Hibernate* some custom restriction are checked, e.g. having a unique symbol name within one project. To reduce the amount of queries to the database, *Hibernate* uses a method called "lazy-loading", which means that it only fetches the data currently needed from the database. This is a great way to improve performance, but leads into some problems when this behaviour is combined with other frameworks. To ensure that all data, that should be send through the REST interface are actually received from the database, it is required to force *Hibernate* to load more data, which is also taken care of in the DAO layer [17].
It is worth mentioning that all *Action* types are stored in only one table. This simplified the table layout in the database and is in most cases faster than working with multiple tables. Property names are reused when possible to minimize the amount of columns within the table, which also reduces the amount of *null* values.

4.1.2. (De-)Serializing an Entity

Similar to the way *Hibernate* handles the entities to store them into a database, *Jackson* can create a valid JSON from it. Per default setting *Jackson* uses the property name to create the name used in the JSON. This can be changed with the "@JsonProperty"

```
1  public class SymbolDAO {
2    Session session;
3
4    /* Initializing the Hibernate session. */
5
6    public void create(Symbol symbol) {
7      session.save(symbol);
8    }
9
10   public List<Symbol> getAll(Long projectId) {
11     return session.createCriteria(Symbol.class)
12                   .add(Restrictions.eq("project.id", projectId))
13                   .list();
14   }
15 }
```

Listing 4.2: A Sample DAO for the Sample Entity Class.

annotation and it is possible to exclude a field from the JSON using the "@JsonIgnore" (line 5 in listing 4.1) annotation. This annotations can not only be applied to properties but also to the getter and setter methods. This way read only properties can be created. Listing 4.3 shows the created JSON of the sample *Symbol* entity (listing 4.1). *Jackson* also allows to read JSON using the same mapping rules.

4.2. Creating the REST Endpoints

Besides using a database and to be able to handle JSON documents, the actual REST API is another core feature of this thesis.

Listing 4.4 shows a small example of a REST resource in Java. In the REST resources annotation are also important. To be recognized as an endpoint the resource class must have the "@Path" annotation (line 1). The value of this annotation determines the URL, here "../projects/{project_id}/symbols". The part within the curly braces ('{' and '}') is a parameter within the url, like the ID of a project and can be used in the methods. Every method that should be a endpoint must have an annotation about the implemented HTTP method, like "@GET" (line 6). Also it is possible to extend the path of the class, so actually the path for the "get" method is "../projects/{project_id}/symbols/{symbols_id}". Again

```
1 {
2   "project": { ... },
3   "name": "..."
4 }
```

Listing 4.3: JSON of the Sample Entity.

with another parameter in the path. Those parameters are received form the function via the "@PathParam" annotation (line 8, 9) on actual method parameters.
Aside form the to the annotation for a REST interface, the "@Inject" annotation (line 3) is used here. This allows to inject an DAOs into that class without any real instantiation through a constructor or setter. This concept is one of the key concepts from the Java EE environment where the server would take care an provide a pool of DAOs, which are injected when necessary. In . The used server software *Jetty* is not a full application server and has no build in support for the injection feature, so in ALEX this injection is done by *HK2*[1] provided by *Jersey*. This reduced the scalability and could potential yield problems when the application is deployed to full application server that support this feature, but it was the easiest and second best option.
To make the REST resources actually work, it is necessary to tell the server which classes to use and under which URL these classes should be available. This configuration can be done in a file called "web.xml", which stores the general configuration of a Java web application. Another way to configure the server is to create a special *Application* class as entry point for the application, which does the configuration within the source code. The last approach is used in ALEX to use the code injection feature.

4.3. Connection to the LearnLib

The *LearnLib* provides is the core of the learning part in ALEX.
As of right now ALEX supports the Extended L*, Discrimination Tree, DHC and TTT algorithm implemented in the *LearnLib*. To allow a more detailed view on the internal data structures, some of them are also stored and send out when a learning result is requested. This includes the observation table if the L* algorithm was used or the Discrimination

[1] `https://hk2.java.net`

```
1  @Path("/project/{project_id}/symbols")
2  public class SymbolResource {
3      @Inject
4      private SymbolDAO symbolDAO;
5
6      @GET
7      @Path("/{symbol_id}")
8      public Response get(@PathParam("project_id") Long projectId,
9                          @PathParam("symbol_id") Long symbolId) {
10         Symbol symb = symbolDAO.get(porjectId, symbolId);
11         return Response.ok(symb).build();
12     }
13 }
```

Listing 4.4: A Sample Resource

Tree of the Discrimination Tree algorithm. Also basic support for the data structures of the TTT algorithm are implemented. It is obviously possible to get the hypotheses and furthermore the discrimination tree is also stored. So that the trie is the only missing part, because sadly there was no easy way to receive it directly out of the algorithm instance.
Implemented is a JSON proxy for the random word, complete and the sample set oracle. A sample set oracle allows the user to specify one or more counterexamples, which will be verify against the SUL with the help of membership queries. Using the same method it is also possible to receive the output of a (possible) counterexample by just providing the input symbols.
Writing a "wrapper" around the Compact Mealy because the *LearnLib* is not design to be store in a database or to have a nice JSON format. The wrapper takes care of this and can be created from any Mealy Machine form the *AutomataLib*[2]. Because nearly every algorithm has its own implementation of a Mealy Machine, which fits the algorithm needs, the given automata will be converted into the representation of a compact mealy with only integers as states. This leads to a consistent format of states and transitions. Furthermore it is possible to use new Mealy Machine types introduced with the new algorithms without touching this part of ALEX. The conversion is done by giving every state from the *AutamataLib* Mealy Machine an unique integer as identifier within the new *CompactMealy*. Afterwards the transitions of the new *CompactMelay* are created by iterating over the provided transitions and adding new transitions with the corresponding integers. The input and output will just be copied. The only downside of this is that there is no guarantee that the nodes have a "nice" order. For example the state with the number "2" could be the initial node and the state "0" could not be directly reached form there and node "1" could be actually the third node to reach.
The *LearnLib* framework has a some predefined filters included. One of them is a simple cache to remember queries that were already executed. So if a query is requested again the cache can answer directly without asking the SUL which leads to a performance gain. Please also note that the cache can only answer a complete query and not just parts of it because the whole query has to be performed on the SUL to get the correct result.
Extracting the statistics of a learning process is done by using some standard Java techniques and different filters provided by the *LearnLib*:

Duration Keep track of the time before entering the learn loop and take the time after each iteration. This difference is then the duration. This also gives access to the start time.

#MQs To count the amount of membership queries the *ResetCounterSUL* is used. This SUL increments a counter in the *pre()* method befor delegating the method call down to another SUL

[2] *AutomataLib* is used in the *LearnLib* to represent the different machines, like a mealy machine.

#EQs The amount of created equivalence queries is counted by a counter which is incremented each time the *findCounterexample(hyp, alphabet)* of the equivalence oracle is called.

Symbols Executed The *SymbolCounterSUL* is used to count the total amount of symbols called during the learning. This is done by adding the size of each query send to SUL to a sum before delegating the method call down to another SUL.

4.4. Actions and Connectors

The design specifies that different ways to interact with a web application should work together, i.e. every *Symbol* could have *Actions* that have totally different meaning and thus implementation.
To achieve this every *Symbol* has an *execute* method, which receives *ConnectorManager* (see figure 4.1 parameter and which is called by the learn algorithm. This method calls then all the actions of the *Symbol* in the right order. Every *Action* has an *execute* method itself and can ask the provided *ConnectorManager* for a certain connector (e.g. *WebSiteConnector*, *WebServiceConnecctor*, ...). Those specialized connectors know how to interact with a web front-end, a REST API or a variable storage. The *Action* is also specials on one task an can perform its tasks with these connectors (see figure 4.2).
Using additional abstract classes, like the *WebSymbolAction* class, we can customize the process of getting a connector and all subclasses of the *WebSymbolAction* just need to have an *execute* method which expects directly a *WebSiteConnector*.
The *WebSiteConnector* uses *Selenium* to simulate the user interaction with a browser front-end of a web application. The *WebServiceConnector* uses *Jersey* as a client for the REST API of the SUL. To have a consistence data storage to exchange information between them, *Counters* and *Variables* are used. *Counters* are simple integer values, that can be set or simply incremented. To allow new learning processes without any reset of the SUL *Counters* persisted with Hibernate, too. An the other hand *Variables* have a smaller scope of only one learning process and thus a simple *HashMap* is used as storage.

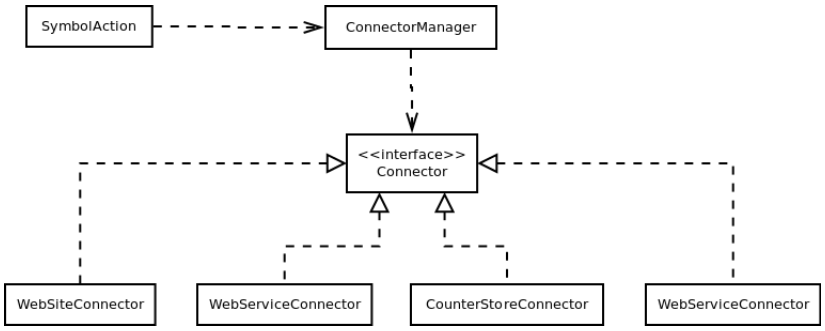

Figure 4.1.: UML Class Diagram for the Connector.

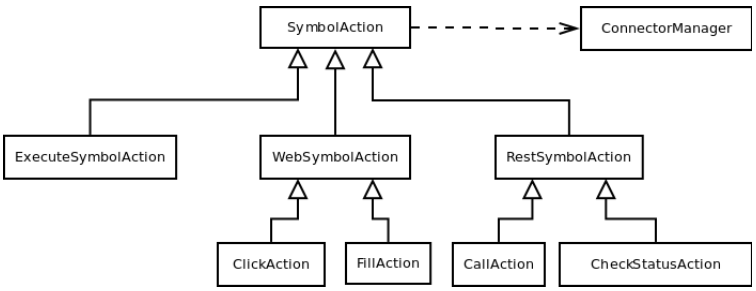

Figure 4.2.: A Simplified UML diagram on the action layer.

4.5. Extendable Architecture

The possibility to extend ALEX is one of the goals specified under section 1.1.
With respect to this this goal ALEX uses the multi module project feature of *Maven*, which makes the structure of the project cleaner. Within the main module three child modules are defined:

API This is the smallest module and only contains some annotations and important interfaces.

Processors The processors module provides annotations processor, which can be called during the compilation of Java source code. Those processors allow code generation with FreeMarker[3].

Main This module holds the real application and uses the other tow packages to be expandable. When doing a clean checkout of the code it may seems that some classes are missing, but actually those classes are generated through the processors.

To process the annotations at compile time is a complex process and requires more effort than to pares them during the runtime. But creating the code at compile time has better performance and is more secure. One example of the generated code is that entities, which should be persisted through Hibernate, will be found trough the "@Entity" annotation, so we create the list of entries at compile time and do not have to manually keep track of them. Another example is the list of algorithms that ALEX can use.

4.6. A Standalone Server

One goal is that the site should be as standalone as possible. So it would be a good way to not only provide a .war file, but also a .jar, which can be executed to start a simple, local server. This small server is created using *Jetty* and the *Jetty Embedded* project. Within a few lines of code the necessary configurations are done and the *.war file deployed. Per

[3] `http://freemarker.org/`

default the server runs on port 8000 to prevent conflicts with other *Jetty* instances, but this can be configure over a command line argument, e.g. "-port=8080".

4.7. Providing Documentation

Beside the previous sections it is necessary to provide a good readable and easy to understand description of the API.

To provide this documentation *Swagger*[4] is used, which defines a way to describe a REST API through a JSON formatted specification. Additionally the *Swagger UI*[5] allows to display this information directly in the browser and also provides the possible to test the described REST interface directly from there (see figure 4.3).

While this is quite comfortable it leads to the problem that when the API code is changed, the documentation must be updated, too. This is additional workload and it would not take long until the real API and the documentation differ. To prevent this the *Swagger Doclet*[6] is used. This is an extension for *JavaDoc* which collects all the information *Swagger* needs directly from the source code and form some additional tags within the JavaDoc and can be executed at compile time.

[4] `http://swagger.io`
[5] `https://github.com/swagger-api/swagger-ui`
[6] `https://github.com/teamcarma/swagger-jaxrs-doclet`

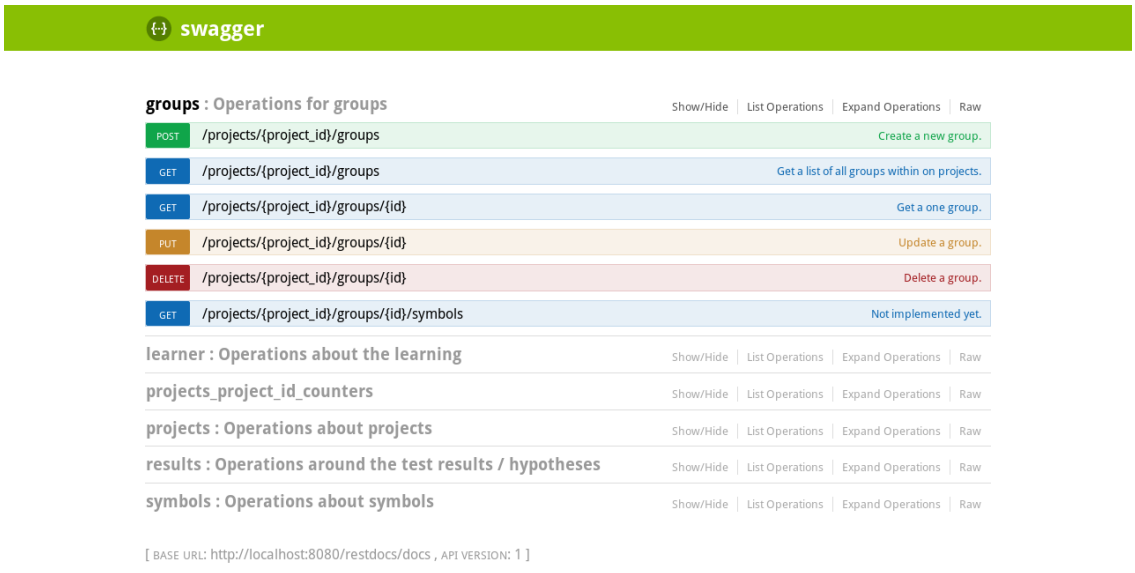

Figure 4.3.: Glance on the Swagger UI

Because the Swagger has some problems with inheritance of entities and to provide more general information about the project, only relying on Swagger is not enough. So the "site" feature of *Maven* is used to provide further documentation. This is enhanced with different Most of the documentation could be written in Markdown[7]. To provide a custom look the *Reflow Maven Skin*[8] is used.

4.8. Testing

While ALEX is a testing tool, it is not possible to test it right away with itself. So different and more common approaches are used. It is important to make sure that this project is working as intended, because if the testing tool has a bug it can cause bigger troubles, from overlooking errors to calling bugs, where everything is working OK. However ALEX relies on the user input and if e.g. the reset symbol does not do an actual reset, ALEX must not have an way to to recognize this. So it is even more important to check for error carefully, so that it is easy for the user to debug his settings.

The project is manly tested using unit test, which are written before the actual production code. This development process is called Test Driven Development (TDD) and additionally Behaviour Driven Development (BDD) [6] is used in this project. Behind BDD is the idea to test only one of expected behaviours of a component per test case. Therefore it recommends a naming convention for the test cases ("should..." or "ensureThat..."). Every test is then structured in three parts: **Given** a set of preconditions and **when** some method is called or code executed, **then** assert the parameter that should have changed. To do the unit test the popular testing framework *JUnit*[9] is used. The duration of these tests should be a small as possible because they should run as often as possible to verify the correct state of the project.

To decrease the runtime one possible way is to use mocks to test the layers independently. The DAO layer can only be tested with the real entities, but the *Actions* in the entity layer should not have to call much logic. The REST interface however is tested against mocks which simulate the DAOs behaviour without testing on a database. This improves test performance and makes sure that the presentation layer work against the business logic definition and not only with one implementation.

To test the actual REST implementation the Jersey testing frame work is used. This framework allows to run the production code within a small server, without dealing to much with the set-up. It integrates well in the JUnit test by offering a "runner" in which a test class is executed. This runner hides the server set-up and configuration. The Jersey

[7] `https://daringfireball.net/projects/markdown`
[8] `http://andriusvelykis.github.io/reflow-maven-skin/skin/themes`
[9] `http://junit.org`

testing framework offers support for different server software and when using a lightweight one, the performance is acceptable for unit test.
To test the whole project a real learning process should be simulated. Most some parts of the unit test use mock and running a learn process within the unit test would make their runtime to long. So a bunch of integration tests were implemented which run on static HTML sites and a very small REST application to verify the project.
Testing ALEX with it self leads into different problems. Firstly this could only be done if the project has reached a certain level of features and complexity, because it is necessary to start a learning process to learn the application by itself. This leads to the point, where it would be easy to test e.g. CRUD operation on the different data sets, but this would be a boring task. It would be more interesting if we could learn how ALEX does a learning process on the *ToDo-App*. This however would take a lot of time because a learning process on the *ToDo-App* takes up to several minutes, even if we would reduce the amount of input symbols. Also this time is not constant and a wait action with a long duration would be needed to wait of the and of the learning process. In the end doing this recursive learning would take a very long amount of time.

4.9. An IFrame Proxy

Last but not least a resource was implemented which is not closely related to the learning set up. The front-end devolved by Alexander Bainczyk has the focus on creating the symbols in an easy, graphics based way. To achieve this a picker for HTML elements was created. This picker shows the website in question to test within the front end and it is possible to select one element for further use. Showing another website from a different domain causes some problems because of some cross domain restrictions in JavaScript. To workaround these restriction a proxy was introduced. This proxy passes get and post request through to let the front end browser believe that everything comes form just one domain. While doing so all links and references were changed to have the proxy url as prefix, so that following links and other references in the page, will cause the new request to use the proxy, too. Lastly cookies which are send to or received from the site behind the proxy are send through as well.

5. Evaluation

This chapter takes a look at the *ToDo-App* created by Alexander Bainczyk and me and how it was originally tested with Active Automata Learning. After this a new testing attempt with the service-oriented interface will be done, so that in the end of this chapter a comparison between those approaches can be done.

5.1. Old Tests of the ToDo-App

The first Active Automata Learning based testing of the *ToDo-App* tried to create a model with user registration, login and logout and CRUD operations on task (figure 5.1). This was realized by writing Java code using frameworks like the *LearnLib* or *Selenium*. Overall the web interface was tested with 28 different symbols, because every symbol had only one action. This decision was made to ensure that the used technologies were working together as intended. This included many read actions, which created reflexive transition at every state and every symbols would only be OK at only one state and would otherwise fail. This helped to identify the meaning of every state but led to a huge and unreadable model. Also it took quite some time to learn this amount of symbols. The reset was done in both cases by using counting to have a new user on every membership query, this could be done because, every user has his own tasks, so creating a new user and working with his tasks has no effect on other users and their tasks. A hard reset, like clearing the database, was not implemented in the *ToDo-App*. The web interface and the web service of the *ToDo-App* could not be learned together and the web service part was tested with fewer symbols, because it was needed to receive an apikey form the web interface to authenticate over the REST API to do a reset. The result was printed in the command line and an image of the model was rendered. So it there was no persistence between every test run and it could happen that old model images were overwritten. Also at this time no statistics were written down and only a rough estimation of the duration for each test was available. Because of the unreadable models a simple model checker was implemented, which was configured by different annotations on the source code of each symbol.

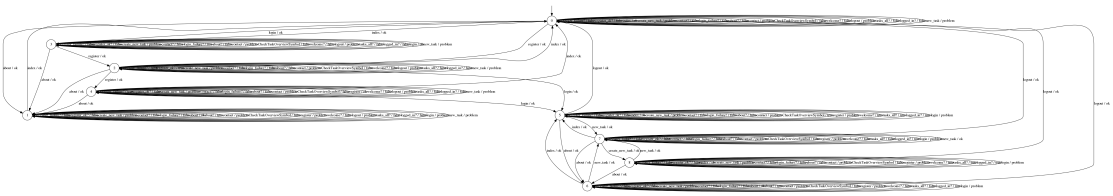

Figure 5.1.: Final hypotheses of the *ToDo-App* created through the old approach

5.2. A New Test of the ToDo-App using ALEX

In this case study only some simple symbols for CRUD operations on tasks are created. The creation of a new tasks is limited, so that only one tasks per query can be created. This is done because tasks have no unique property and otherwise an infinite amount of tasks could be created. While this is not a big problem itself, because only a certain depth/ length of words is tested, this causes issues when we want to delete a task. In that case we could only delete how many tasks were created previously. To limit the amount of tasks is an easy way around that. Also this allows to store the id of the tasks in a variable called "taskid", which makes e.g. the update symbol easier, because it can refer directly to the the task. The update symbol just changes the name of the task and to test, if an update was successful or not, two read symbols are defined. The first one just test if a task was created and the second one checks, if the task is updated. Those read symbols are also executed within the create symbol to create the limitation. This CRUD operations are defined for the web and the REST interface.

To reset the SUL one reset symbol is used. This symbol deletes all remaining aspects of a previous run, i.e. it log the user out, resets some variables and goes to the start page of the *ToDo-App*. Afterwards it creates a new user and start a session with him. It also creates a valid API key, which is stored in a variable called "apikey" and will be used by the REST symbols. As described in chapter 2 it is faster to not do a hard reset of the database, which even with only a few symbols Also it is possible take a look at the created data afterwards and validate the actions executed by the symbols.

This symbols are created through the REST interface of ALEX using the Swagger UI (see listing A.1). The API documentation provided through this tool is useful and even offers some auto completion. The biggest problem is, that the actions are not included in this overview. After defining all symbols it was possible to take a look at them from the front-end created by Alexander Biancyk and it is possible to start a learning process through other way, like *curl*, an *Unix* command-line tool to create HTTP requests. Imagining the created models only through the JSON format is not easy, but it was possible to look at them and all the statistics through the front-end, too. The persistence of counters allows to run different learning processes quickly without any interaction with the running *ToDo-App* instance, so some test runs are done to see the possibilities of ALEX.

Figures 5.2 and 5.3 show the final models for set ups which only used the web site interface or the the web service interface. Both were created using the L* algorithm and a random word equivalence query (min length = 1, max length = 10, max no. of word = 20). Both models are representing the *ToDo-App* as expected. For a human eye it is easy to see, that both mealy machine are isomorph.

It is also possible to run both symbol groups together and the result is also as expected. It is easy to see that the CRUD operation over both interfaces have the same effect on the *ToDo-App*. By the nature of the learning process and the fact, that we are using multiple ways to communicate with the SUL, which can have all sorts of side effects, this is not a formal proof that the *ToDo-App* is bug free.

Those tests are done with different settings to compare different algorithms and equivalence oracles. The first two runs were done using the random word equivalence oracle with a minimum word length of 1, a maximum word length of 10 and a maximum amount of word of 20. The algorithms were L* for Mealy Machines (table 5.1) and the newer TTT algorithm (table 5.2).

It is interesting to see that running both symbol types together, which doubles the amount of input symbols, must not have a huge impact on the duration of the test. In some cases running both together was faster than the sum of both individual test runs. This could be caused by the sequence the algorithm is using them. If the algorithm has a prefix two web symbols and tries other symbols out, this web symbols are called over and over again. This is slower than using two REST symbols. So we in some cases using REST and web together can cause the REST symbols to boost the test of web symbols. Also using the random word with the same parameters could be problematic, because doubeling the size of Σ while not changing the length of words or the amount of created words could mean a loss of accuracy.

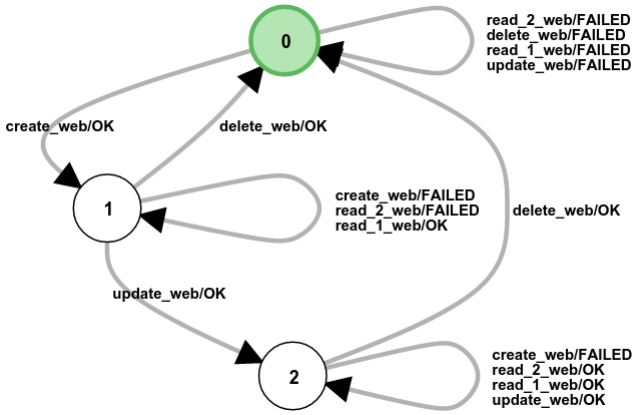

Figure 5.2.: Final Hypotheses of the ToDo-App for a learning process with only web symbols

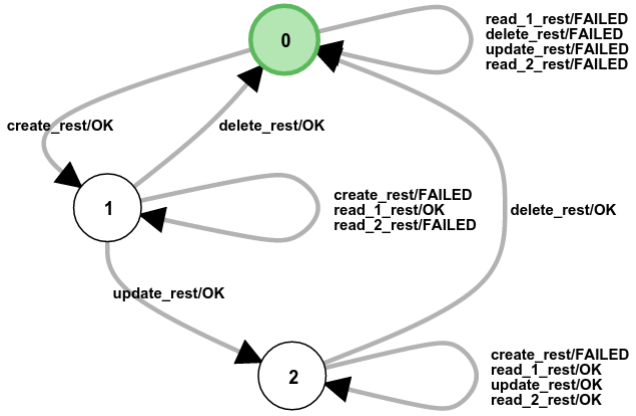

Figure 5.3.: Final Hypotheses of the ToDo-App for a learning process with only REST symbols

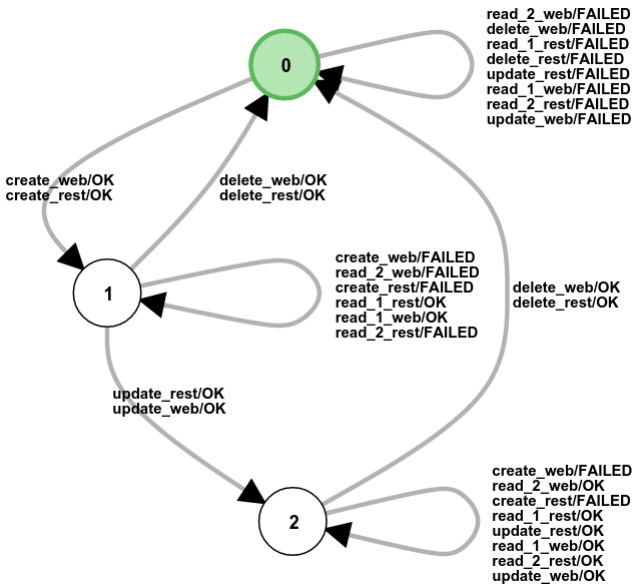

Figure 5.4.: Final Hypotheses of the ToDo-App for a mixed learning process

Data	Web Symbols	Rest Symbols	Both Together
Duration (rounded)	123s	20s	233s
#MQs	115	115	360
#Symbols Called	368	368	1078
#EQs	1	1	1

Table 5.1.: Statistic with the extended L*, using Random Word (min-length: 1, max-length: 10, #words: 20)

Data	Web Symbols	Rest Symbols	Both Together
Duration (rounded)	70s	10s	57s
#MQs	121	75	185
#Symbols Called	467	268	589
#EQs	6	3	6

Table 5.2.: Statistic with TTT, using Random Word (min-length: 1, max-length: 10, #words: 20)

The last two test used the same algorithm, but to find the counterexamples a complete oracle is was used. The minimum depth was set to 1 and the maximum depth was 3 (tables 5.3 & 5.4).
Exploring the whole model to a specific depth takes more time and the larger amount of input symbols has a bigger effect on the runtime than in the previous tests. On the other hand it is clear that our model is up to the specified depth correct.

5.3. Comparing the Different Approaches

A direct comparison of the created models is not possible because of the different symbols and concepts behind both of the approaches and would only help to talk about the *ToDo-App* and not about ALEX. However it is possible to compare the process behind them. Writing the JSON definitions took some time to get used to it, but it is faster than to write the learn code by hand. The persistence of old models and the various statistics recorded by ALEX are interesting to see and allow easy comparison of different algorithm. The model checking implement in the first approach helps to decide if the final hypotheses fulfills some constrains and thus if the ToDo-App seems to be implemented correctly. In ALEX every model has to be validated by hand and if the models grow bigger it could be handy to have also some model checking in ALEX. Reading those model in JSON This is not a specific problem of this project but is located in the purely text based approach of a service-oriented interface. Using the front-end of Alexander Bainczyk made things here a bit more comfortable.

Data	Web Symbols	Rest Symbols	Both Together
Duration (rounded)	291s	32s	1408s
#MQs	250	250	1450
#Symbols Called	690	690	4180
#EQs	1	1	1

Table 5.3.: Statistic with the extended L*, using Complete (min. depth: 1, max. depth: 3)

Data	Web Symbols	Rest Symbols	Both Together
Duration (rounded)	173s	29s	1388s
#MQs	279	534	1818
#Symbols Called	739	1413	4938
#EQs	3	6	6

Table 5.4.: Statistic with TTT, using Complete (min. depth: 1, max. depth: 3)

6. Conclusion

Lets take a look back to chapter 1 where the goals of this theses were defined and decide whether they are met or not:

Language independent The developed service-oriented interface is only build on HTTP request and the JSON format. It is not necessary to know any existing programming language and interacting with the SUL is also possible without further frameworks or tools.

Persistence The Symbols and Learn Results are stored in a database. This allows to take a look at different test runs and with the recorded statistics it is easy to compare different algorithms.

Standalone The created service-oriented interface can be deployed to every Jetty server, a lightweight server written in Java. The installation process of a Jetty server is straight forward and well documented. But it is not even required for the user to set up a own server instance, because some a small standalone application based on Jetty was created. The service-oriented interface can be used with simple tools for HTTP request or the *Swagger UI*. The set up depends not on any front-end.

Well documented A good API documentation is provided through Swagger and this documentation is updated with the source code. The simple UI for Swagger offers a good visual representation and it is possible to try . Besides Swagger additional documentation is provided through the site feature of *Maven*. This documentation is not complete and it would be nice to have more information for users.

Extendable This goal was not in the main focus but some parts of ALEX are still design with this in mind. For example it should be fairly easy to add new algorithm into the service-oriented interface.

Furthermore it is possible to use different techniques to interact with a web application within one symbol, which creates new opportunities.

6.1. Further Improvements

Like most software projects ALEX has reached a point, where it has a plenty of features, but could still be improved. There a plenty of features and ideas that might be interesting to evaluate and eventual to implement.
Firstly it would be nice to visualize more internal data structures of the algorithms. This would help to improve the understanding of the different algorithms and includes the trie structure of the TTT algorithm, which is missing right now because there was no straight forward way of getting it out of the *LearnLib* and into a nice JSON format.
A bigger change would be to allow symbols to have more outputs possible outputs. Every symbols contains a set of actions, which are executed sequentially to determine the output of the symbol. This output can currently only be "OK" or "Failed". A flexible way to realize this could be to break the sequential execution of actions to a more control flow oriented way. Each Action could still be executed successfully or failed, but depending of that the next action could be selected. This would turn into a (binary) action tree (see figure 6.1). At the leaves an outcome of the symbol could be defined. Also things like returning the last status code could be a handy feature.
To allow more action types it maybe also needed to create more connectors. This could include an interface to talk to a database or a SOAP interface. This implies also support for more data formats. Currently only the JSON format is supported, but the XML format is also pretty common.
Besides the simple cache it could be possible to add support for the reuse filter, which could allow even more performance increments. A database is already used and could store counter and variable values of each state in it. Additionally further information, that the reuse filter can be used for more complex reuse cases (e.g. read only actions), could be added to the action informations.
While the expandable architecture works already well in some parts in the back end, an integration of the front end should be considered. Defining a user friendly name for

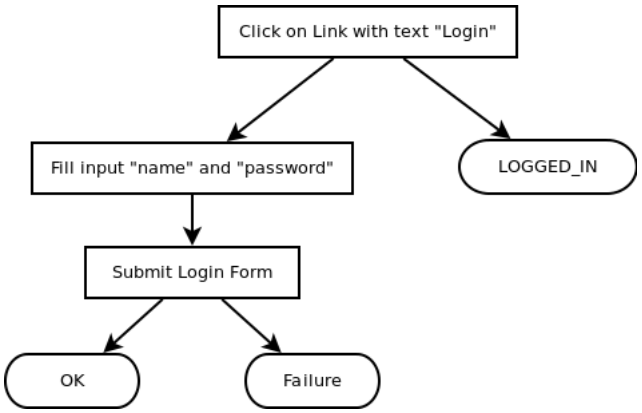

Figure 6.1.: Possible configuration of a control flow based symbol.

algorithms that the front end could use directly on the algorithm class and not to need to edit yet another file would make extending ALEX pretty fun. Also it would maybe be possible to provide some sort of connection between an action definition in the back end and a template in the front end, so that adding new action types to the project could be done with minimal effort.
Next it could be a good idea to introduce a connection between two symbols, that are equal. This would allow some simple model checking on the created hypotheses, instead of verifying by hand that two equal symbols have the same translation and output behaviour. To consider this it would be handy to get some user feedback if the mixed learning is done relatively often. Maybe this knowledge could also be used to improve algorithms by design, the way counterexamples are searched or to increase the efficiency of the reuse filter.
Talking about model checkers, the different ways to have a way to check the learned models continuously, e.g. shown by Stephan Windmüller [23], would allow to include ALEX in a continues building workflow, where a new version could be rejected if the new model differs from previous ones.
Lastly it could be considered to add a multi user management and a authentication procedure. This would ALEX to be securely used outside of a close local network and could reduce the running instances of ALEX where on developer maybe is not allowed to test certain applications.

6.2. User Feedback

ALEX will be used in the summer 2015 iteration of Webtechnologien II as a tool for the students This opens the opportunity to hear real user experience, e.g. through a user experience evaluation. Together with some students form the last year a comparison between the old approach and how ALEX changed this is possible. After all, these user reports should be the true unit of measurement here.

A. Further Information

```
{
  "name": "Update Task (Web)",
  "abbreviation": "update_web",
  "actions": [
    {
      "type": "web_goto",
      "negated": false,
      "ignoreFailure": false,
      "url": "/tasks/show/{{$taskid}}"
    },
    {
      "type": "web_click",
      "negated": false,
      "ignoreFailure": false,
      "node": "HTML > BODY:nth-child(2) > DIV:nth-child(3) > DIV:nth-
          child(1) > DIV:nth-child(2) > DIV:nth-child(8) > A:nth-
          child(1)"
    },
    {
      "type": "web_fill",
      "negated": false,
      "ignoreFailure": false,
      "node": "#title",
      "value": "Test Task Updated"
    },
    {
      "type": "web_click",
      "negated": false,
      "ignoreFailure": false,
      "node": "#task > DIV:nth-child(8) > DIV:nth-child(1) > BUTTON:
          nth-child(1)"
    }
  ]
}
```

Listing A.1: Symbol to Update a Task through the web interface.

Type	Name	Description
Web	CheckNode	Check if a certain element is present on the website.
Web	CheckText	Check if a certain text is part of the website body.
Web	ClearAktion	Clear an input field.
Web	Click	Click on an element.
Web	Fill	Clear and fill an input field with some text.
Web	Goto	Request a specific site.
Web	Submit	Submit a form.
Web	Select	Select an option form an select input field.
REST	Call	Do a REST Call.
REST	CheckAttributeExists	Check if the response has an specific attribute.
REST	CheckAttributeType	Check if an attribute in the response has a specific type.
REST	CheckAttributeValue	Check if an attribute in the response has a specific value.
REST	CheckHeaderField	Check if the response has a certain header field.
REST	CheckStatus	Check if a previous response returned the expected HTTP. status.
REST	CheckText	Check the response body.
Misc	SetCounter	Set a counter to a new value.
Misc	IncrementCounter	Increment a counter.
Misc	SetVariable	Set a variable to a new value.
Misc	SetVariableByHTMLElement	Set a variable to a value form a website element.
Misc	SetVariableByJSONAttribute	Set a variable to a value form a JSON response.
Misc.	Wait	Wait for a specific amount of time. This can be useful for background tasks or AJAX calls, but should be used with care because it can slow the learning process down.
Misc.	Execute Symbol	Include and execute another symbol.

Table A.1.: Overview about the actions

List of Acronyms

ALEX *Automata Learning Experience*

API Application Programmer Interface

BDD Behaviour Driven Development

CRUD Create, Read, Update and Delete

DAO Data Access Object

HTTP Hypertext Transfer Protocol

ID Identifier

JPA Java Persistence API

Java EE Java Enterprise Edition

JSON JavaScript Object Notation

OCS Online Conference System

ORM Object Relation Mapper

POJO Plain Old Java Object

REST Representational State Transfer

SUL System under Learning

TDD Test Driven Development

XML Extensible Markup Language

List of Figures

Bibliography

[1] Dana Angluin. Learning regular sets from queries and counterexamples. In *Information and Computation 75*, pages 87–106. 1987.

[2] Alexander Bainczyk. Simplicity-oriented web-based control of active automata learning. Bachelor thesis, TU Dortmund University, 2015.

[3] Oliver Bauer, Johannes Neubauer, Bernhard Steffen, and Falk Howar. Reusing system states by active learning algorithms. In *Eternal Systems*, pages 61–78. Springer, 2012.

[4] Facebook Developers. *Using the Graph API, Version 2.3*, 2015. `https://developers.facebook.com/docs/graph-api/using-graph-api/v2.3`.

[5] Roy Thomas Fielding. *Architectural styles and the design of network-based software architectures*. PhD thesis, University of California, Irvine, 2000.

[6] Aslak Hellesoy and Matt Wynne. *The Cucumber Book - Behaviour-Driven Development for Testers and Developers*. The Pragmatic Programmers, 2012.

[7] "Twitch Interactive Inc.". *Twitch API v3 - Errors*, 2015. `https://github.com/justintv/twitch-api`.

[8] "Twitter Inc.". *Error Codes & Responses HTTP Status Codes*, 2015. `https://dev.twitter.com/overview/api/response-codes`.

[9] Malte Isberner, Falk Howar, and Bernhard Steffen. The ttt algorithm: A redundancy-free approach to active automata learning. In *Runtime Verification*, pages 307–322. Springer, 2014.

[10] Malte Isberner and Bernhard Steffen. An abstract framework for counterexample analysis in active automata learning. In *Proc. ICGI*, 2014.

[11] Madhusudhan Konda. *Just Hibernate*. " O'Reilly Media, Inc.", 2014.

[12] Tiziana Margaria, Oliver Niese, Harald Raffelt, and Bernhard 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. IEEE, 2004.

[13] Mark Masse. *REST API design rulebook*. " O'Reilly Media, Inc.", 2011.

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

[15] Maik Merten, Bernhard Steffen, Falk Howar, and Tiziana Margaria. Next generation learnlib. In *Tools and Algorithms for the Construction and Analysis of Systems - 17th International Conference, TACAS 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26-April 3, 2011. Proceedings*, pages 220–223, 2011.

[16] Nurzhan Nurseitov, Michael Paulson, Randall Reynolds, and Clemente Izurieta. Comparison of json and xml data interchange formats: A case study. *Caine*, 2009:157–162, 2009.

[17] Inc.“ ”Red Hat. *HIBERNATE - Relational Persistence for Idiomatic Java*, 2015. `http://docs.jboss.org/hibernate/orm/4.3/manual/en-US/html`.

[18] Leonard Richardson and Sam Ruby. *RESTful web services.* “ O'Reilly Media, Inc.”, 2008.

[19] The Internet Society. *Hypertext Transfer Protocol – HTTP/1.1*, 1999. `http://tools.ietf.org/html/rfc2616`.

[20] The Internet Society. *The application/json Media Type for JavaScript Object Notation (JSON)*, 2006. `https://tools.ietf.org/html/rfc4627`.

[21] Bernhard Steffen, Falk Howar, and Maik Merten. Introduction to active automata learning from a practical perspective. In *Formal Methods for Eternal Networked Software Systems*, pages 256–296. Springer, 2011.

[22] Stephan Windmüller. *Kontinuierliche Qualitätskontrolle von Webanwendungen auf Basis maschinengelernter Modelle.* PhD thesis, TU Dortmund University, Germany, 2014.

[23] Stephan Windmüller, Johannes Neubauer, Bernhard Steffen, Falk Howar, and Oliver Bauer. Active continuous quality control. In *Proceedings of the 16th International ACM Sigsoft Symposium on Component-based Software Engineering*, CBSE '13, pages 111–120. ACM, 2013.