ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Information Security
www.infsec.ethz.ch

Stefan Hildenbrand

Generation of Test Cases

Programming a tool for the generation of test cases from finite automata

Semester Thesis
Summer Semester 2005
ETH Zürich, October 10th, 2005

Supervisor: Diana Senn
Professor: David Basin

Zusammenfassung

Wir entwickeln ein Tool zur Generierung von Testfällen von Endlichen Automaten. Das Tool erlaubt dem User einen Endlichen Automaten zu spezifizieren und generiert anschliessend mit Hilfe der Wp-Methode die Testfälle. Weitere Methoden k“onnen einfach hinzugef“ugt werden.

Wir konzentrieren die Entwicklung auf den Einsatz im Zusammenhang mit Netzwerkprotokollen, daher bietet das Tool die Möglichkeit eigene Eingabe- und Ausgabesymbole zu definieren.

Abstract

We develop a tool for test case generation from finite state machines. The tool allows the user to specify a finite state machine and then generates the test cases using the Wp-method. Other methods can be easily added.

We focus the development on network protocols therefore the tool offers the possibility to define input and output symbols.

Contents

List of Figures

Part I.
Introduction and Theory

This part covers the basics needed for this project such as some theory about finite state machines and the most important terms used in this project.

1. About Finite State Machines

1.1. Finite State Machines

In the world of computer science, basically everything can be represented as a so called finite state machine (FSM). Such FSMs are composed of states, transitions and actions. A state stores information about the past, i.e. it reflects the input changes from the system start to the present moment. A transition indicates a state change and is described by a condition that would need to be fulfilled to enable the transition. An action is a description of an activity that is to be performed at a given moment.[var]

1.2. Mealy Machines

Since FSMs are quite important to computer science, there has been a lot of research on that topic. Different types of FSMs are known. In this thesis, we concentrate on so called Mealy Machines. This type of FSM extends the base type with an output function.

Formally a Mealy Machine can be described as a 7-tuple: $\mathcal{A} = (Q, \Sigma, \Omega, \delta, \lambda, q_0, F)$ [var].

- Q is a finite set of states. $|Q| < \infty$
- Σ is the input alphabet. $|\Sigma| < \infty$
- Ω is the output alphabet. $|\Omega| < \infty$
- δ is the transition function. $\delta : Q \times \Sigma \to Q$
- λ is the output function $\lambda : Q \times \Sigma \to \Omega$
- $q_0 \in Q$ is the initial state.
- $F \subseteq Q$ is a (finite) set of possible accepting states. If the FSM stops in a state in F after reading input $w \in \Sigma^*$, then w is part of the language $L(A)$.

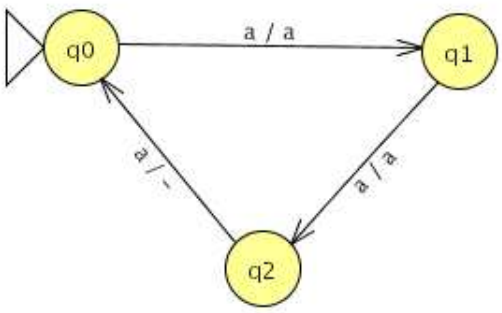


Figure 1: a very simple Mealy Machine, which ignores every third input

1.3. FSMs and Computer Science

In computer science, FSMs are used to model a certain behaviour of software, hardware or other processes, for example a TCP network protocol. A device (meaning a computer or a firewall or the like) reacts usually on inputs from its environment and changes its state accordingly. Quite commonly it produces an output, too. Exactly this behaviour is reflected in a Mealy-Machine. Therefore a network protocol (e.g. TCP) can be represented as a FSM. For humans this graphical representation is usually easier to understand than the source code of the actual implementation.

2. About Testing

2.1. Testing

Testing is a process used to investigate the properties of a certain *implementation under test (IUT)*, especially the correctness, completeness and quality. In order to be able to test, one needs to know what correct is. This means one needs a complete specification of the IUT, i.e. a so called *model*. Usually, testing does not guarantee complete correctness, since there can always be cases, which were not covered by the set of test cases (i.e. *test suite*) and sometimes even the model is not complete, giving another source of errors.

2.2. Testing in Computer Science

In computer science, testing used to play a much bigger role than in other engineering disciplines. During the construction of an airplane, the engineers can calculate most of the properties the airplane is going to have, because they have their blue prints, the physical laws and mathematics to their help. Surely, they are doing a test flight before delivering the airplane to the customer, but this is just the last step.

As opposed to this approach, developing computer devices usually goes through many test flights. On one hand it is easier and cheaper to run a test build of software than to fly a prototype of an airplane. On the other hand, in computer science rarely any calculations can be done on the model, since there are usually no blue prints. Although there exist so called formal methods which can be seen as a form of blue prints, coming up with them is often omitted. Bugs in software are widely accepted, a bug in an airplane is not.

2.3. Testing Using FSMs

One possible model of a computer device is an FSM. The expected behaviour is reflected in a FSM. To test the IUT, one checks whether the IUT gets to the same state as the FSM does, after reading the same input sequence. In this case, both the model and the IUT must be FSMs. By using Mealy Machines, one can test whether the IUT produces the same output sequence as the model does, when reading the same input sequence. This allows testing more types of IUT.

Usually a FSM reflecting a computer device has many states and many more transitions. Since testing is a very time consuming and therefore expensive process, one is eager to reduce the number of test cases that need to be run against the IUT. There exist several algorithms

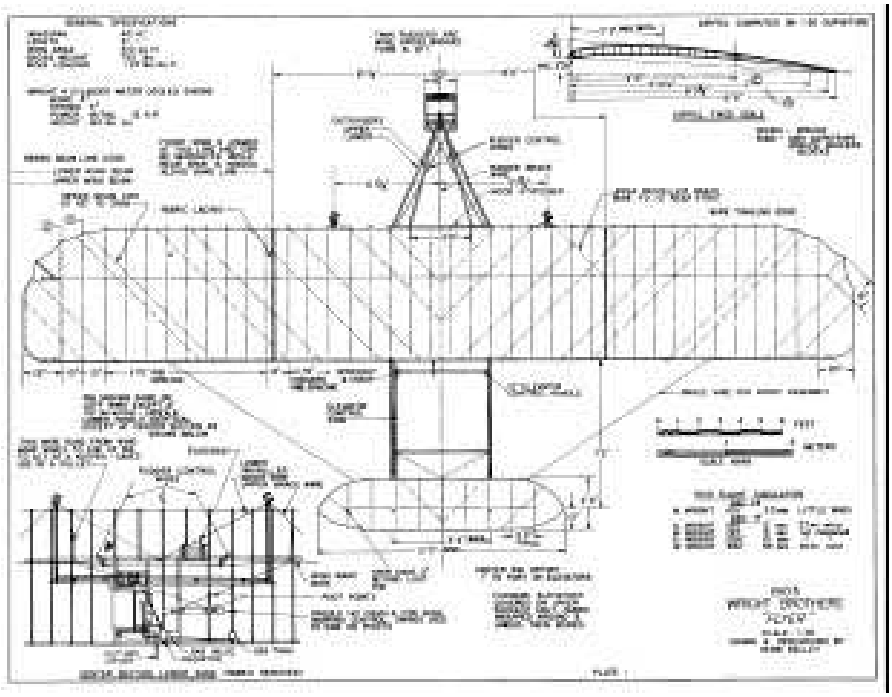


Figure 2: an old blueprint of the Wright Flyer

to extract a test suite from a FSM. The algorithm used in this thesis is explained in the next section.

3. Generating a Test Suite From a FSM

This section is not intended as a complete review of the known methods to generate test cases from FSMs. This is covered in [FvBK+91]. This section is neither supposed to go in deep detail on the presented method, but should give the basics to understand the so called *partial W-method* (Wp-method) used in our tool.

My supervisor decided on this method because the W- and Wp-method are most generally applicable among these methods and the Wp-method was preferred because it yields smaller test suites than the original W-method.

3.1. Some Notations and Definitions

First we introduce some notations and definitions later used to present the algorithm. The same definitions are used in [FvBK+91].

$\boldsymbol{Mi \xrightarrow{x/y} Mj}$ means that the FSM M in state Mi fed with input x responds with an output y and makes the transition to the state Mj .

$\boldsymbol{Mi \xrightarrow{p} Mj}$ means that the FSM M is originally in state Mi and goes to the state Mj when the input sequence p is applied.

Mi|p represents the output sequence produced by M in state Mi when the input sequence p is applied.

V1.V2 The concatenation of two sets $V1$ and $V2$ of input sequences is a set of input sequences defined as follows: $V1.V2 \equiv \{v1.v2 \mid v1 \in V1, v2 \in V2\}$ where $v1.v2$ stands for the concatenation of the two sequences $v1$ and $v2$.

***V*-equivalence of states** Given a set V of input sequences, two states Si and Ik are V-equivalent (written as "$Si \approx_V Ik$") if S in Si and I in Ik respond with identical output sequences to each input sequence in V.

Equivalence of states Two states Si and Ik are equivalent (written as "$Si \approx Ik$") if they are V-equivalent for any set V.

Equivalence of FSMs Two FSMs S and I are equivalent if their initial states So and Io are equivalent.

minimal FSM A FSM M is minimal if the number of states in M is less than or equal to the number of states for any machine M' which is equivalent to M.

state cover set *Q* Let Q be a set of input sequences. Q is a state cover set of S if for each state Si of S there is an input sequence $pi \in q$ such that $So \xrightarrow{pi} Si$. The empty input sequence (ϵ) belongs to Q.

transition cover set *P* Let P be a set of input sequences. P is a transition cover set of S if for each transition $Si \xrightarrow{x/y} Sj$ there are sequences p and $p.x$ in P such that $So \xrightarrow{p} Si$ and $So \xrightarrow{p.x} Sj$. The empty input sequence (ϵ) is a member of P. By definition, each transition cover set P contains a subset which is also a state cover set.

identification set *Wi* A set of input sequences Wi is an identification set of state Si if and only if for each state Sj in S (with $i \neq j$) there exists an input sequence p of Wi such that $Si|p \neq Sj|p$ and no subset of Wi has this property.

characterisation set *W* The union of all identification sets Wi is a characterisation set. A characterisation set consists of input sequences that can distinguish between the behaviours of every pair of states in a minimal automaton.

3.2. The Wp-Method

The Wp-method as described in [FvBK+91] is an improvement of the original W-method as described in [Cho78]. Using the Wp-method reduces the length of the test suite.

The Wp-method makes some assumptions about the specification S and the implementation I. The specification S should be minimal. This is a necessary (and sufficient) condition for the existence of a characterisation set W. In order to guarantee the error-detection power of this method, S and I are assumed to be completely specified and deterministic. Moreover the following explanation assumes that the number of states in S and I are equal. But the more general case when the implementation has more states than the specification can also be handled by this method. This extension is explained in [FvBK+91].

Like several other methods, the Wp-method follows a two-phase approach:

The first phase checks that all the states defined by the specification are identifiable in the implementation, and also checks for each state Ik that it can be identified by the identification set Wk. At the same time, the transitions leading from the initial state to these states are checked for correct output and state transfer. This happens by applying the following test sequences to the IUT: $Q.W$ where Q is a state cover set and W is a set of input sequences including at least all the identification sets Wi of all states.

The second phase checks all transitions which were not checked during the first phase. This is done by applying the sequences of the transition cover set P which are not contained in Q concatenated with the corresponding Wi:

$$R \otimes \mathcal{W} = \bigcup_{p \in R} \{p\}.Wj$$

where $R = P - Q$ and $\mathcal{W}$ is the set of all identification sets. Wj is the identification set of Sj in $\mathcal{W}$ and Sj is reached by p, meaning $So \xrightarrow{p} Sj$.

If the implementation I passes the tests of both phases, it is equivalent to the specification S. More details and a proof of this assertion can be found in [FvBK+91]. A detailed example can be found in Section 10.

3.3. Finding the Identification Sets

For the presented method, it is necessary to have the cover sets and identification sets. Finding the state and transition cover sets is basically a broad search in the automaton, but finding the identification sets is not that easy. Our tool uses the algorithm for the so called "Multiple Preset Diagnosing Experiment" as presented in [Gil62]. The author of this reference distinguishes between *diagnosing* and *homing*, *preset* and *adaptive* as well as *simple* and *multiple* experiments:

> *The diagnosing problem:* It is known that a given machine M, whose transition table is available, is in one of the states $\sigma_{i_1}, \sigma_{i_2}, \ldots, \sigma_{i_m}$. Find this state.
>
> *Preset experiments:* the applied input sequence is completely determined in advance
>
> *Adaptive experiments:* the applied input sequence is composed of two or more subsequences, each subsequence (except the first) determined on the basis of responses resulting from preceding subsequences.
>
> *Simple experiments:* only one copy of the machine is required
>
> *Multiple experiments:* more than one copy of the machine is required

For our needs, we must solve the diagnosing problem. Since this tool is supposed to generate a static test suite, we use the "preset" algorithm, which does not depend on the output the IUT produces.

I decided to use the "multiple" algorithm, because we can have virtual copies of the IUT with the help of the reset transition which must be implemented anyway (this is a requirement for the Wp-method). Additionally, the "multiple preset diagnosing experiment" is more powerful than the simple one, meaning that it can solve some problems the simple one cannot.

This algorithm has one shortcoming though, as described in [Gil62]:

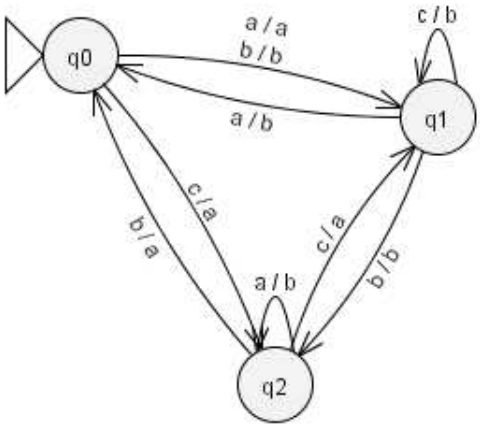
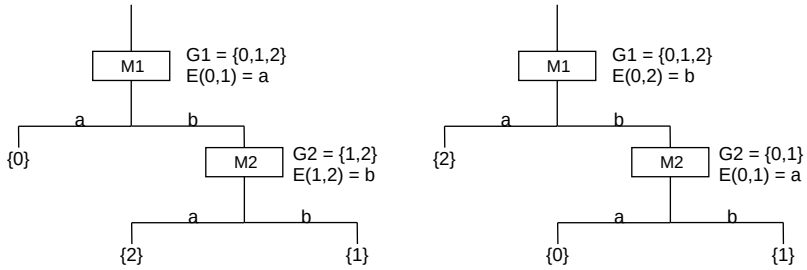


Figure 3: example automaton



(a) MET using 'a' as first input (b) MET using 'b' as first input

Figure 4: Multiple-experiment trees (MET) for example automaton

> It may be noted that, although the design procedure for the multiple experiment can minimize the length of the input sequence applied to each copy, it generally does not minimize the total length of the experiment or its multiplicity.

The selection criteria for the distinguishing sequences (take one of the shortest) applied in the different steps is ambiguous. Sometimes a distinguishing sequence applied in later steps could separate other paths of the diagnosing tree too. Therefore some of the input sequences in the identification sets are superfluous. Therefore the power of this algorithm costs the requirement, that the tool should produce minimal test suites.

This shortcoming is best described in an example taken from [FvBK$^+$91] and presented in Figure 3.

Obviously the identification sets for this automaton are either $\{\{a\}, \{a, b\}, \{b\}\}$ or $\{\{a\}, \{c\}, \{b\}\}$. But none of these can be found by the algorithm presented in [Gil62]. The diagnosing tree looks either like Figure 4(a) yielding $\{\{a\}, \{a, b\}, \{a, b\}\}$ or like Figure 4(b), yielding $\{\{a, b\}, \{a, b\}, \{b\}\}$ as identification sets.

Part II.
Development

In this part the development of the tool is presented: Discussing the requirements, doing a web research for a similar tool which can be extended and the programming work.

4. Requirements

The first step of the development of this tool was to collect the requirements it should fulfil.

4.1. Purpose of the Tool

This tool is supposed to allow the user to draw a graphical representation of a FSM and translating it into a machine readable representation. From this machine readable representation the tool should generate so called *abstract test cases*. An abstract test case consists of an input sequence to be fed to the IUT and an output sequence, which can then be compared to the output sequence the IUT produced. 'Abstract' indicates, that usually these test cases cannot be fed directly to the IUT since the model contains variables in the input sequences which must be instantiated with concrete values. E.g. a model of the TCP-protocol does not contain the actual IP-addresses of sender and receiver but most likely imposes some restrictions on them. Another tool must then be used to instantiate the variables with IP-addresses fitting the restrictions.

4.2. Interfaces

The tool consists of two completely different interfaces. One interface to the user and one to other tools.

4.2.1. Graphical Interface

The graphical interface lets the user construct arbitrary Mealy Machines. This interface needs three views.

Automaton view The Automaton view allows the user to construct the actual automaton. This means to draw states and the transitions between these states. The transitions consist of an input symbol and an output symbol. Different input symbols can lead to the same state and produce the same output.

The symbol is referenced by its name. The actual meaning of the symbol is not of interest here, but must be defined in the Symbol view for later use in test case generation.

Form view The Form view allows the user to edit the base form of the symbols of the input alphabet the FSM should listen to. This view allows to add, edit and delete the fields a symbol can have. A field of a symbol consists of a name and a data type.

Symbol view The Symbol view allows to create, edit and remove symbols to the alphabet. This view basically should implement a mask of the defined form, allowing to fill the fields with values.

4.2.2. Machine Interface

The machine interface is used in different ways. One purpose of this interface is to allow communication with other tools, e.g. to instantiate the test cases generated by this tool. The other purpose is to be able to save and reload the work.

test cases The tool produces a structured output which can be used by other tools (such as fwtest [Zau]).

FSM The tool allows also to save and load the constructed FSM, the form and the symbols.

4.3. Functions

By defining the interfaces, the functions are quite obvious:

- construction of Mealy Machines, including an alphabet consisting of user defined symbols based on a user defined format
- translation of the graphical representation into a machine readable form and vice versa
- generation of abstract test cases from the machine readable form

4.4. Other Requirements

There are a few other requirements which do not fit the above distinction:

- the tool should not impose high requirements on the environment it is run in
- the tool should be released under some form of free license

5. Evaluation

After knowing what the tool should be capable to do, I started looking for other tools which do similar tasks. The goal was to find a tool which was easy to adapt but also powerful enough to satisfy our needs.

5.1. Web Research

Since FSMs are a widely researched topic in computer science there exist lots of tools covering all kind of tasks in this area. But most of the hits I found by googling the net were the opposite approach to the problem. There are lots of toolboxes, libraries and other software out there to display a static graphical representation of a given automaton, but I found it rather difficult to find software which was also offering to construct or edit an automaton in a graphical and interactive way.

In computer science, FSMs are basic knowledge, so looking for some interactive teaching utilities brought two tools into my scope, Exorciser and JFLAP, which are discussed later in this section.

One page I found quite helpful which I would like to mention here, was the link page of the Grail+ Project[1]. Lots of packages and toolboxes concerning FSMs and the like can be found there. A reference to the JFLAP project is listed there as well. Although the descriptions are not always up to date, it is an alternative to crawling through the many hits google produces.

5.2. Exorciser

The Exorciser [ea] is an interactive teaching software covering different topics in theoretical computer science such as finite state machines, regular languages and the like. Different theses have improved this tool since its first presentation in 2002. Offering a pleasing graphical interface and being developed at ETH too, this seemed to me a very promising foundation for my tool.

One of the focuses of the Exorciser was to make the tool easy to extend with new exercises covering other topics in the field of theoretical computer science. But this extendability just worked for completely new exercises. Trying to adapt the FSM part to Mealy Machines seemed rather difficult. Missing comments in the source and an out of date report of the semester thesis [HL01], which added the FSM part to the Exorciser framework made work very hard. The initial commit of the FSM was changed in meanwhile, but not well documented such that several classes mentioned in [HL01] do not even exist anymore. Additionally, in most parts of the project, transition inputs were restricted to a single character, which is not convenient for our project. Changing this restriction seemed heavy work too.

These difficulties encouraged me to look for another software I could use as foundation for my tool.

5.3. JFLAP

JFLAP [Rod] is a package of graphical tools which can be used as an aid in learning the basic concepts of Formal Languages and Automata Theory. This software lets the user create and

[1] `http://www.csd.uwo.ca/research/grail/links.html`

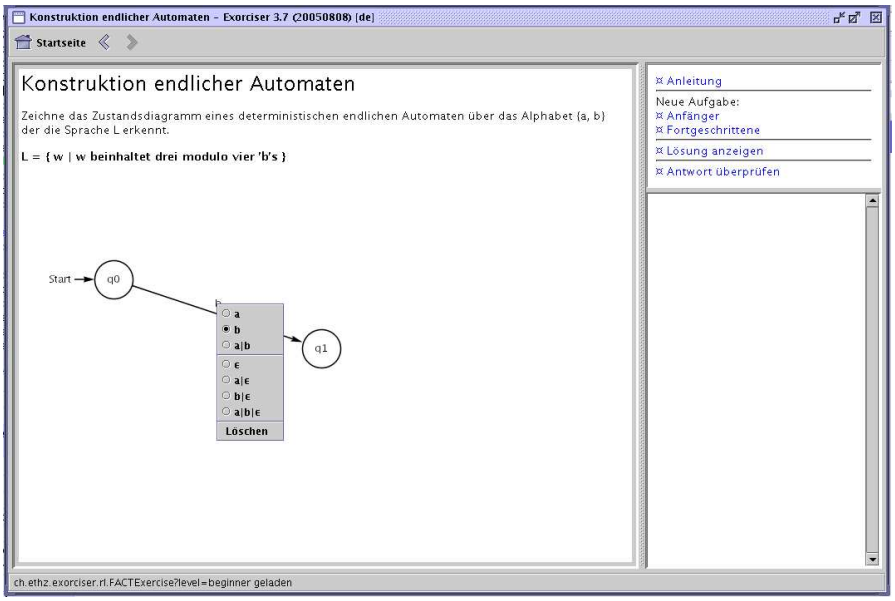


Figure 5: Screenshot of Exorciser: editing a FSM

edit different types of finite state machines and then offers a variety of tasks on this FSM such as converting non-deterministic automata into deterministic ones.

The structure of the program allows to add new types of FSMs by simply implementing an interface. By adding a new type of transition, different kinds of transitions (such as input-output pairs needed for Mealy Machines) can easily be added. The source code is well commented and the software may be reused under the usual restrictions.

With this tool I found the foundation of a graphical representation of FSM which seemed quite simple to extend to the needs of this tool. Providing a graphical user interface and an obvious way to add other types of FSMs together with a simple XML-based way to save and load existing FSMs made this software my choice.

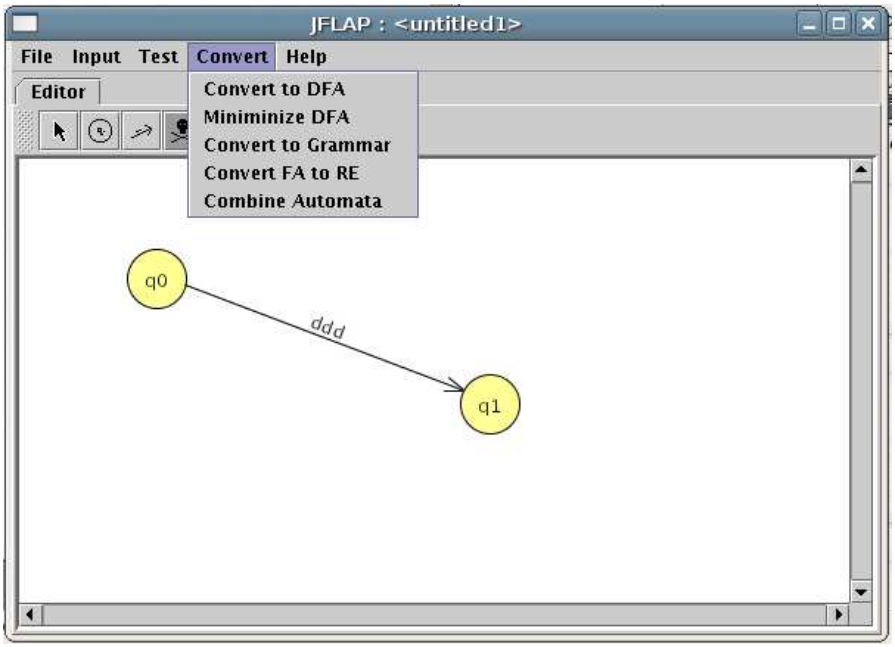


Figure 6: Screenshot of JFLAP: editing a FSM

Figure 7: Screenshot of JFLAP: offering different types of FSMs

```
<!--Created with JFLAP 4.0b14.-->
<structure>
        <type>fa</type>
        <!--The list of states.-->
        <state id="0">
                <x>371.0</x>
                <y>196.0</y>
        </state>
        <state id="1">
                <x>157.0</x>
                <y>204.0</y>
        </state>
        <!--The list of transitions.-->
        <transition>
                <from>0</from>
                <to>1</to>
                <read>a</read>
        </transition>
        <transition>
                <from>1</from>
                <to>0</to>
                <read>a</read>
        </transition>
</structure>
```

Figure 8: XML-Output produced by JFLAP

6. Design

As pointed out before, there are two basic actions our tool has to perform:

1. *Modelling.* Let the user draw an automaton representing a sequence of actions of interest.
2. *Test Case Generation.* Generate test cases for the given automaton.

6.1. Modelling - Capturing the Essence

The first step in modelling is to capture the Essence of the thing one wants to model. The keyword is *Abstraction*. The main purpose of this tool will be to generate test cases for network protocols. Such network protocols can become very complex. Especially the input and output behaviour of the participating hosts can differ from protocol to protocol. So called *packets* are exchanged between the hosts, but the format of TCP-packets has not much in common with the format of IP-packets.

So we need to be able to define the input and output format. But on the other side, for test case generation, the actual meaning of the symbols is not important at all. Therefore one should separate these things.

For complete modelling, we need to define

1. the actual control flow, meaning the automaton with its transitions
2. the form of the input and output symbols
3. the input and output symbols themselves

The abstraction of the actual meaning of the symbols is quite easy. In the automaton, the symbols are referred with a key unique for each symbol. During test case generation, this key is sufficient. When it comes to using the test cases, these keys can be filled with meaningful information by using the symbol definitions.

For the format of the symbols, which consists of several fields, it seems to be enough to have a name (used to identify a field) and a data type for each field. How the actual test packet is constructed from this field must be determined otherwise. Most likely not all fields of the actual test packet are of interest. Reviewing some of the most important network protocols, I decided to go with the following data types: Integer, String, Character, Boolean, and Variable. The special type of *Variable* is introduced to distinguish between a defined String from a placeholder which has to be replaced with an actual value.

6.2. Test Case Generation - Combining the Algorithms

The Generation of Test Cases following the Wp-method is performed through several steps. We need to get from a graphical representation of an automaton to a test suite.

1. *Cover sets.* We need the state and transition cover sets. These sets can be found by performing a broad search in the automaton.
2. *minimal distinguishing sequences.* Since we want to apply the algorithm to find the identification sets as presented in [Gil62] we first need to find the minimal distinguishing sequences. I proceeded as suggested in Section 4.4 of [Gil62].

3. *identification sets.* Section 4.10 of [Gil62] presents an algorithm to find the identification sets.

4. *generate test cases for the states.* Now knowing all necessary sets we can perform phase 1 of the Wp-method.

5. *generate test cases for the transitions.* The next step is then to perform phase 2 of the Wp-method.

6. *completing the test cases.* So far we generated input sequences needed to test the automaton. Now we feed these sequences to the modelled automaton and note the expected output for each input sequence. This gives then a complete test case consisting of an input sequence and an output sequence which has to be compared to the output the IUT generates.

6.3. File Format - Presenting the Results

6.3.1. File Format for the Automaton

The file format for saving the automaton must be some kind of structured file. Therefore a XML-file seems suitable. The following informations have to be written to the file:

- type of the automaton (in our case always Mealy Machine, but could be subject to change)
- reference to the alphabet this automaton is using
- list of states
- list of transitions

For each state we need to save:

- id
- position
- flags for initial and final

For each transition we need to save:

- start and end state
- input and output symbol

These elements can be directly transformed to a XML-file. An example is shown in Figure 9.

```
<?xml version="1.0" encoding="UTF-8"?>
<structure>
        <type>mm</type>
        <alphabet>alpha1</alphabet>
        <!--The list of states.-->
        <state id="0">
                <x>130.0</x>
                <y>53.0</y>
                <initial/>
        </state>
        <state id="1">
                <x>362.0</x>
                <y>57.0</y>
        </state>
        <!--The list of transitions.-->
        <transition>
                <from>0</from>
                <to>0</to>
                <read>a</read>
                <write>a</write>
        </transition>
        <transition>
                <from>1</from>
                <to>0</to>
                <read>a</read>
                <write>a</write>
        </transition>
</structure>
```

Figure 9: File format for automaton

```
<?xml version="1.0" encoding="UTF-8"?>
<structure>
        <type>alphabet</type>
        <name>alpha1</name>
        <!--Following the fields of the form-->
        <field type="0">field1</field>
        <!--Following the symbols-->
        <symbol name="symbol1">
                <value>456</value>
        </symbol>
</structure>
```

Figure 10: File format for alphabets

```
(b/b)(a/b)(a/a)(a/b)(a/a)
(b/b)(a/b)(a/a)
(a/a)(a/a)
```

Figure 11: File format for test suite output

6.3.2. File Format for the Alphabet

The same list of important fields can be collected for alphabets:

- name of the alphabet
- list of fields each symbol must have
- list of symbols

For each field we need to know:

- name
- data type

For each field we must save:

- name
- the values for each field defined in the alphabet

This list of elements can as well be quickly transformed in a XML-file as the example in Figure 10 shows.

6.3.3. File Format for the Test Cases

The output is then dumped to a file in the format as shown in Figure 11:

- one test case per row
- each step of the automaton is enclosed in `()`
- input and output symbol are separated by `/`

7. Implementation

7.1. Adaption of JFLAP

7.1.1. Extending JFLAP for Mealy Machines

The design of JFLAP includes an interface called 'Automaton', that all different types of automaton extend. It was easy to adapt the Turing Machine implementation, which is part of the JFLAP package, to represent a Mealy Machine.

7.1.2. Adding More Editor Capabilities to JFLAP

The editor of JFLAP is built up from several panes which are collected in an environment. These elements are predefined in the Java base library. To add the possibility to edit the alphabet and its symbols needed for our tool, I just had to add new panes to the environment. The editor panes and the objects they present to the user are connected using the Observer-pattern [var], so changes in the alphabet or its symbols are immediately represented in the editor views.

7.1.3. XML Output

For this step JFLAP was perfectly designed, too. The generation of XML output is kind of modularized in JFLAP such that transducers for new types of automaton and new types of objects (like the additional alphabet I added to a mealy machine) can easily be added. Several helper functions predefined in an abstract super class take care of most of the XML specific details.

7.2. Implementation of the Test Case Generation Algorithms

7.2.1. Transforming the Graphical Representation to the Transition Table View

The rather unhandy distinction between states and transitions in JFLAP had to be translated to a format easier to handle. The states in the design of JFLAP do not know anything about the transitions around them. This is fine for graphical representation, but for actually working with the automaton, this seemed not the right way. The obvious way were transition tables, since they are the starting point in [Gil62], too.

I implemented the transition table as an array of so called transition table entries. Each of these entries represented a state and contained the next state and the output symbol for each symbol in the alphabet. This information could be gathered from the JFLAP-view of the automaton by inspecting all transitions and filling in the entries step by step.

7.2.2. Generating Minimal Distinguishing Sequences Using P-Tables

For constructing the P-Tables as described in [Gil62] I had to come up with another object type. I decided to use a top-down approach:

- A *P-Table* has to know what equivalence classes it contains. Also it knows in which equivalence class all states are. This is kept in the P-Table because otherwise each row would have to be updated if a state gets into a new equivalence class.

- A *P-Table-Entry* then represents an equivalence class. It knows which states it contains.
- A *P-Table-Row* finally represents one state. It knows about the next states on each symbol.

Now implementing the algorithm of Gill was not difficult any more: Take the P-Table (which knows the annotations), and for each P-Table-Entry in that P-Table, distribute the P-Table-Rows in new P-Table-Entries and add them to the original P-Table. When all Entries are inspected, restart until each Entry contains only one row.

During the work at this algorithm, I noticed that this algorithm can be applied incrementally and we do not need to save the intermediate steps. All we need to do is to generate the minimal distinguishing sequences (MDS) after we completed a step. A MDS between two states can be constructed by prepending a symbol which causes the automaton to go into two distinguishable states (meaning they are not in the same equivalence group) to the MDS of these two states.

7.2.3. Finding the Cover Sets by a Broad Search in the Automaton

For this step, the JFLAP representation is as good as the transition table view. We are looking for the shortest path from the initial state to all states and to all transitions.

I implemented this with a queue serving as todo list. This list is initialised with the initial state of the automaton. As long as there are states in the todo list, each state reachable from the first one in the list by one transition and not yet visited is enqueued in the todo list. During this loop all states and transitions are visited. When first visiting a state or transition I record the sequence that led to that object.

7.2.4. Constructing the Multiple-experiment tree

The construction of the Multiple-experiment trees (MET) is recursive. We initialise the procedure by creating a MET with all states in the *admissible set*. An admissible set contains the states the automaton could be in. During the process, the admissible set becomes smaller, since we narrow the set of possibilities.

The constructor looks for the shortest MDS between the states in the admissible set. The output produced when feeding the found MDS to the automaton in all states of the admissible set is then compared and for each subset with the same output, a new MET is constructed with the smaller subset. The construction ends, if there is just one element in the admissible set.

By using a static variable of the Java programming language, each MET can access the same variable. Therefore the identification sets can be constructed incrementally. Each time the tree branches, the used MDS is added to the identification sets of all states in the admissible set. When the recursion ends, the identification sets can be found in that static variable.

7.2.5. Combining the Found Sequences to Test Cases

The combination of the found sequences to test cases is a simple loop trough all elements which have to be concatenated as described in [FvBK+91].

7.2.6. Preparing the Test Cases for Output

After having constructed the test cases, the found input sequences are fed to the automaton and the expected output together with the input sequence is dumped to a file in the format presented earlier.

Part III.
Results

This part presents the results this thesis acquired: a short description of the tool developed and some examples what it can do.

8. Manual

The tool comes with an integrated help function, which I reused from JFLAP. It is a simple HTML-Browser which opens an HTML-document associated with the actual view shown in the application window. I adapted the help pages to reflect the changes I made to JFLAP. Following the description that also appears on the corresponding HTML-page in the tool.

8.1. The Automaton Editor

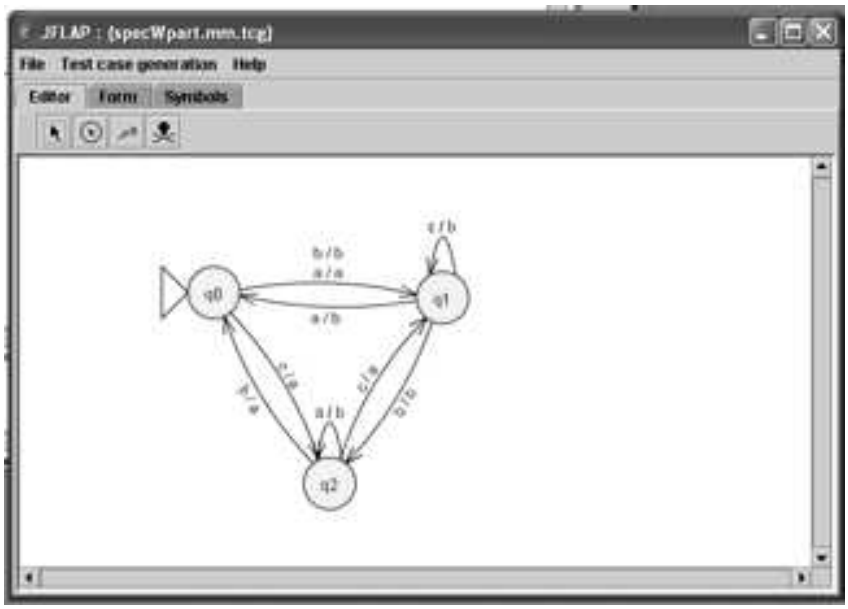


Figure 12: Screenshot: Editing a finite automaton

The Editor Pane

The editor pane has three main components. On the top is a detachable tool bar (the section with the four iconic buttons), and on the bottom right is the canvas where the automaton is drawn. If you defined an alphabet for the automaton, a list of the symbols you defined is on the bottom left.

The automaton is edited through clicks on the canvas. The current action taken in response to those clicks depends on the currently selected tool, which is indicated by a darker background on the icon.

To select another tool, click on it, or use its shortcut key. You can get the shortcut key, as well as a short description of the tool, by holding the cursor over the tool button. After a short time this should display a tool-tip with information.

The Tools

The Attribute Tool (the "A" key)

This tool is used to modify existing states and transitions. To move states, drag them with this tool. Dragging transitions will move transitions. Clicking on a transition edits a transition. A user may right-click (control click on Macs that have only one mouse button) on a state to bring up a pop-up menu to define the state as final, to define the state as initial, to set its label (an auxiliary description for a state), to clear its label (if it has one), or to clear all labels of all states.

The State Tool (the "S" key)

While the state tool is selected, a click on the canvas will create a new state centred at the point that was clicked.

The Transition Tool (the "T" key)

By dragging from one state to another, a transition from the first state clicked to the second state dragged to will be created. Looping transitions from a state to itself are handled the same. Once two states are joined with the tool, the user will then be asked for certain parameters of the transitions.

The Delete Tool (the "D" key)

This will erase any transition or state it is clicked on. If a state is deleted, those transitions incident on or emanating from a state will be deleted as well.

Right-clicking in a blank portion of the canvas will bring up a different pop-up menu. One option of this menu is to hide or show labels; if labels are hidden, the user may hold the mouse cursor over a state to bring up a tool-tip with the label for that state. The other option is the graph layout algorithm, which will rearrange the states of the automaton. As a nota bene, this popup menu is available in all views with an automaton; however, as one might expect the user may use the graph layout algorithm only where JFLAP allows the moving of states.

Editing Transitions

Only one transition may be edited at a time. While a transition is being edited, fields appear that allow the user to change parameters regarding that particular transition. Each field corresponds to a particular parameter of that transition. In order to find out which is which during runtime, hold the mouse cursor over one of the fields; a tool tip will pop up to tell what it is.

A Mealy Machine's transition editor will have two fields, one for the input symbol and one for the output symbol.

To stop editing, take some other action with the tool, or click in an empty space on the canvas, or press return/enter. To cancel the editing of a transition and to revert that transition to its state before the edit, press escape. Pressing shift-return/enter will stop editing the transition, and begin the process of editing a new one between the start and end states of the transition that we just stopped editing.

8.2. The Form Editor

The Form Pane

The Form Pane consists of several input fields. The top one holds the name of the actual alphabet. The ones below represent the fields of this alphabet. Each field has a name and a

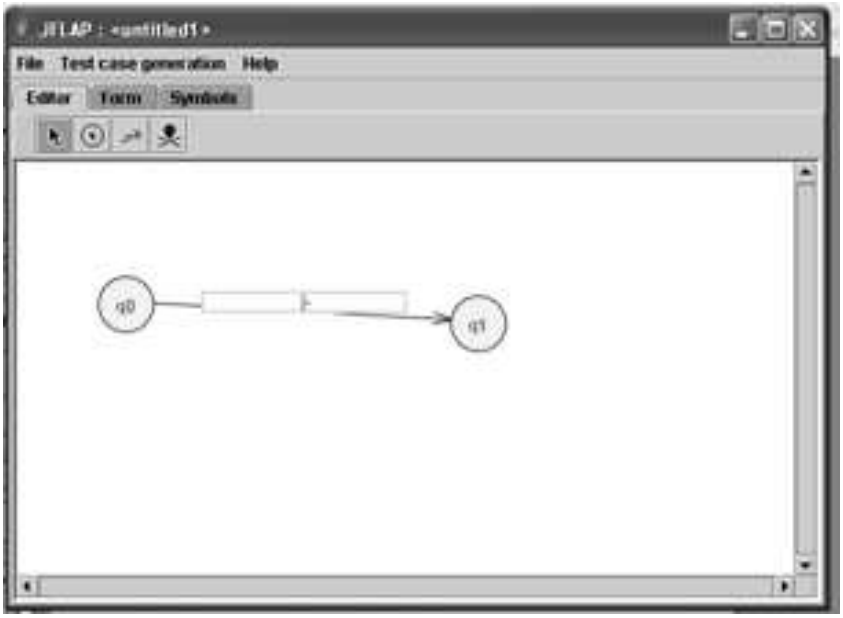


Figure 13: Screenshot: Editing a transition

Figure 14: Screenshot: View Showing the Form Editor

data type.

To add a new field click on the “add field”-button at the bottom. To delete fields, tick them and click on the “delete fields”-button at the bottom.

8.3. The Symbol Editor

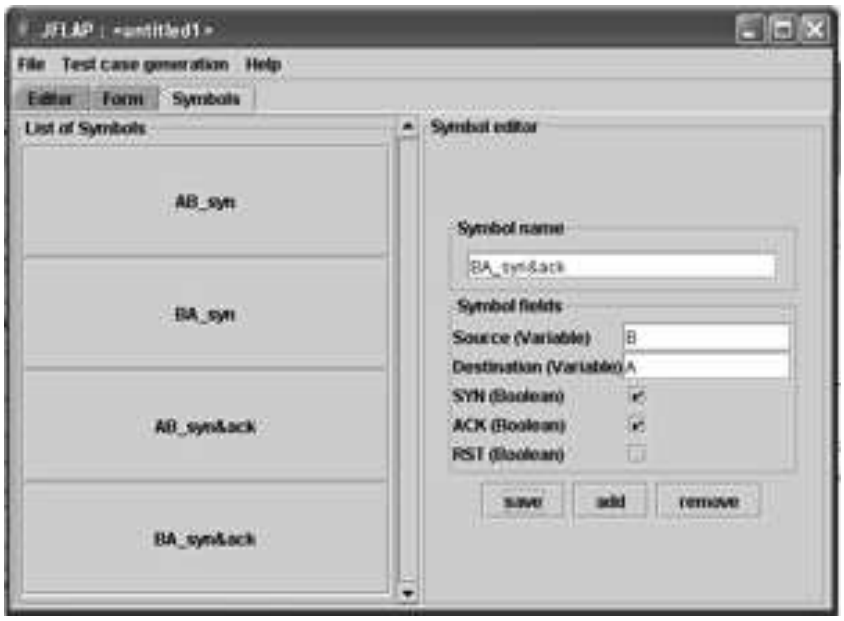


Figure 15: Screenshot: View Showing the Symbol Editor

The Symbol Pane

The Symbol Pane has two main components. On the left is a list of the defined symbols. On the right is a mask to create new and edit defined symbols.

To create a new symbol fill in the fields on the right side (Name on top, the other fields according to the form you previously defined in the Form Editor). When you are finished, click on "save".

To edit a symbol, click on its name on the left side and alter the fields in the editor, when finished, click on "save". To delete a symbol, load it in the editor (by clicking its name on the left side) and click on "delete".

9. Developer Guide

This section should give some hints where to start when extending this tool. More topics are covered in Section 7: Implementation. Section 10.4 might give some information, too.

9.1. Adding Other Algorithms

The package *tcg* contains the classes used to generate the test cases. It has an abstract class called *TestCaseGenerator* which new classes implementing another test case generation method should extend. The extending class must implement the method *generateTestCases* which is supposed to return a set of input sequences which form a test suite. The other

methods of the abstract class then take care of the output to these sequences and dump the complete test suite to a file.

The class *TCGwithTT* can be used as a foundation for new algorithms, too. It extends the *TestCaseGenerator* class by offering a transition table view of the automaton.

To add the new test case generator to the menu of the GUI, one has to add a corresponding line to gui.menu.MenuBarCreator#getTcgMenu.

As another option, the XML-file produced by the tool can be used as a starting point for test case generation. This could be an interface to another application, e.g. if one wants to escape the Java environment.

9.2. Good to Know

The delimiter for separating the symbol names in input sequences is defined in tcg.model.MealyMachineModel#DELIMITER. It is set to ":" by default.

The data type "set" is not implemented. This is just a placeholder for further development.

10. Example

To illustrate the whole process, we go step by step trough a simple example. The example is taken from [FvBK+91]. The same example was used before and is illustrated in Figure 3.

10.1. Drawing the Automaton

The user interface of JFLAP is quite instructive. To draw our example automaton, we select the state tool and click three times in the white space of the editor window. Each time we click, a state is created at this position. We distribute the three states over the pane to have enough space for the transitions.

Next, we add the transitions between the states. We chose the transition tool and click and hold on the state the first transition starts from. We drag the mouse (while still keeping the button down) to the end state of this transition. Then we release the button and an input field for the transition symbols appears. In the left field, we enter the input symbol, in the right field, we enter the output symbol. If we want to use an alphabet with defined symbols, we have to define them first (see below). After having defined the symbols, we can use the buttons on the left in the Editor view to add symbols to the transitions.

To create transitions leading to the same state, just click on that state. Finally we need to define the initial state. This happens by selecting the Attribute tool and right clicking on the desired initial state. A popup menu appears. One of the offered menu items is "initial". We select that item and a large triangle is added to the drawing, indicating the initial state.

10.2. Defining Alphabet and Symbols

For the actual test case generation, the alphabet and symbol functionality is not needed. The unique keys used in the automaton are sufficient. The process of defining an alphabet and symbols for an automaton should be quite instructive and can be read in the online help.

10.3. Starting the test case generation process

When we are finished with drawing our automaton, we select the item "Wp method" from the "test case generation" menu. Now everything runs in background. When test case generation is finished, we are prompted for a file name, where the test suite is saved to.

10.4. Looking behind the scene

Since we are interested in the details of this tool, we take a look behind the scene and show what is going on before we can save the output to a file.

Using the debug mode of eclipse, we inspect the key steps in the process:

```
public Set generateTestCases() {
        generateCoverSets();
        generateIdentificationSets();
        [...]
        result.addAll(generatePhase1());
        result.addAll(generatePhase2());
        return result; }
```

After the first method call above, the variable state cover set contains `[a,c,ϵ]` and the variable transition cover set contains `[c:b, a:b, a:a, a:c, a, c, c:a, c:c, b]`. This result is different from what [FvBK+91] says. But a closer look shows, that the tool just used `a` instead of `b` in the state cover set, which is completely correct, since both `a` and `b` take the automaton to the same state, starting from the initial state. This different choice is then also reflected in the transition cover set, where the first `b` in the multiple input sequences is replaced with `a`.

The next step abstracts the whole process of building up the P-Tables and the multiple-experiment tree. After the execution of that row, the variable identification sets reads: `[[a, b], [a, b], [b]]`. Now this is obviously different from `[[a], [a, b], [b]]` as suggested by [FvBK+91], and there is no way to make it look the same. The reason of this problem is the shortcoming of the used algorithm as described in 3.3. The identification set of one state is a little to large. The consequences of this problem will become apparent in a few steps.

After some intermediate steps, the Wp-method takes control. The test sequences generated in phase 1 are `[c:b, a:b, a:a, a, c:a, b]`. [FvBK+91] comes to `[c:b, b:b, b:a, a, c:a, b]`. The produced sequences differ from the expected ones by the same replacement as described in the cover sets.

Now for the phase 2 results, we get:
`[a:a:a, c:c:b, b:a, a:b:b, a:c:a, c:b:b, c:c:a, c:a:b, b:b, a:c:b, c:b:a, a:a:b]`
Compared to the results from [FvBK+91]:
`[b:a:a, c:c:b, a:a, b:b:b, b:c:a, c:b:a, c:c:a, c:a:b, a:b, b:c:b]`

Here too, the first `a` is replaced with a `b` and vice versa. This should be the same test suite. But now we can see what influence the larger identification set has. The tool produced two more test cases then necessary, because it used more test cases to be sure that the automaton is in the correct state. The output produced by this example is shown in Figure 16.

I injected the state and transition cover sets from [FvBK+91] in the code to see if really the same test suite would be produced and it was the case. Surely the larger identification set caused the same problem. Injecting the shorter identification set finally allowed the tool to generate the exact identical test suite as in [FvBK+91].

```
(a/a)(a/a)(a/a)
(b/b)(b/b)(a/b)(a/a)(a/b)
(b/b)(a/b)(a/a)(a/b)(a/a)(b/b)(a/b)(a/a)(a/b)
(b/b)(a/b)(b/b)(a/a)(a/a)
(b/b)(a/b)(a/a)(a/b)
(b/b)(a/b)(a/a)(b/b)(a/b)(a/a)(a/a)
(b/b)(a/b)(a/a)(a/b)(a/a)(a/a)(a/a)
(b/b)(a/b)(a/a)(a/b)(a/a)(a/a)
(b/b)(a/b)(b/b)(a/a)(a/a)(a/a)
(b/b)(a/b)(a/a)(a/b)(a/a)
(b/b)(a/b)(a/a)
(a/a)(a/a)
(b/b)(a/b)(b/b)(b/b)(a/b)(a/a)(a/b)
(b/b)(a/b)(a/a)(a/b)(b/b)(a/b)(a/a)(a/a)
(a/a)(b/b)(a/b)(a/a)(a/b)
```

Figure 16: Output produced by TCGTool

10.5. Large Example

After this simple and understandable example follows a larger example. This is about TCP connection handling. It shows the results produced by TCGTool when operating on a larger automaton. This example also includes an alphabet. Figure 17 shows a simplified view of the automaton, Figure 18 shows an extract from the XML-file used to store the alphabet and the symbols and Figure 19 shows an extract of the generated test case file.

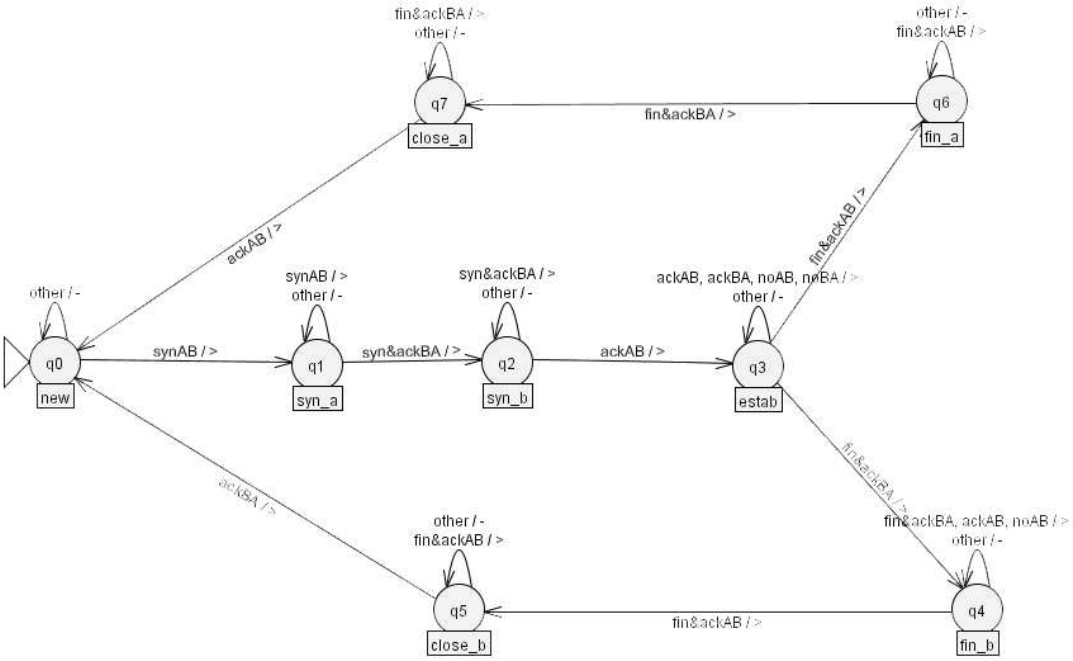


Figure 17: Example 2: Automaton

```
<?xml version="1.0" encoding="UTF-8"?>
<structure>
        <type>alphabet</type>
        <name>TCPAlphabet</name>
        <!--Following the fields of the form-->
        <field type="5">Source</field>
        <field type="5">Destination</field>
        <field type="3">rst</field>
        <field type="3">syn</field>
        <field type="3">ack</field>
        <field type="3">fin</field>
        <!--Following the symbols-->
        <symbol name="finAB">
                <value>A</value>
                <value>B</value>
                <value>false</value>
                <value>false</value>
                <value>false</value>
                <value>true</value>
        </symbol>
        <symbol name="ackBA">
                <value>B</value>
                <value>A</value>
                <value>false</value>
                <value>false</value>
                <value>true</value>
                <value>false</value>
        </symbol>
        <symbol name="ackAB">
                <value>A</value>
                <value>B</value>
                <value>false</value>
                <value>false</value>
                <value>true</value>
                <value>false</value>
        </symbol>
```

Figure 18: Example 2: Extract from the alphabet file

```
(synAB/synAB)(syn&ackBA/syn&ackBA)(ackAB/ackAB)(noAB/noAB)(ackAB/
    ackAB)
(synAB/synAB)(syn&ackBA/syn&ackBA)(ackAB/ackAB)(fin&ackAB/fin&ackAB
    )(fin&ackBA/fin&ackBA)(fin&ackAB/-)
(synAB/synAB)(syn&ackBA/syn&ackBA)(syn&ackAB/-)(ackBA/-)
(synAB/synAB)(syn&ackBA/syn&ackBA)(ackAB/ackAB)(fin&ackAB/fin&ackAB
    )(fin&ackBA/fin&ackBA)(synBA/-)(fin&ackBA/fin&ackBA)
(synAB/synAB)(syn&ackBA/syn&ackBA)(ackAB/ackAB)(fin&ackBA/fin&ackBA
    )(fin&ackAB/fin&ackAB)(fin&ackBA/-)
(synAB/synAB)(syn&ackBA/syn&ackBA)(ackAB/ackAB)(fin&ackBA/fin&ackBA
    )(finAB/-)(ackAB/ackAB)
(synAB/synAB)(syn&ackBA/syn&ackBA)(ackAB/ackAB)(fin&ackBA/fin&ackBA
    )(fin&ackAB/fin&ackAB)(finAB/-)(ackAB/-)
(synAB/synAB)(syn&ackBA/syn&ackBA)(fin&ackBA/-)(fin&ackBA/-)
(synAB/synAB)(syn&ackBA/syn&ackBA)(ackAB/ackAB)(fin&ackAB/fin&ackAB
    )(fin&ackBA/fin&ackBA)(finBA/-)(fin&ackBA/fin&ackBA)
(synAB/synAB)(syn&ackBA/syn&ackBA)(ackAB/ackAB)(fin&ackBA/fin&ackBA
    )(finBA/-)(ackAB/ackAB)
(synAB/synAB)(syn&ackBA/syn&ackBA)(ackAB/ackAB)(fin&ackAB/fin&ackAB
    )(noBA/-)(ackBA/-)
(synAB/synAB)(noAB/-)(ackAB/-)
(synAB/synAB)(syn&ackBA/syn&ackBA)(ackAB/ackAB)(fin&ackAB/fin&ackAB
    )(fin&ackBA/fin&ackBA)(synAB/-)(ackBA/-)
(synAB/synAB)(noBA/-)(ackAB/-)
(synAB/synAB)(syn&ackBA/syn&ackBA)(ackAB/ackAB)(ackAB/ackAB)(ackAB/
    ackAB)
(synAB/synAB)(ackAB/-)(syn&ackBA/syn&ackBA)
(synAB/synAB)(fin&ackBA/-)(syn&ackBA/syn&ackBA)
(synAB/synAB)(syn&ackBA/syn&ackBA)(ackAB/ackAB)(fin&ackBA/fin&ackBA
    )(fin&ackAB/fin&ackAB)(ackAB/-)(ackBA/ackBA)
(synAB/synAB)(syn&ackBA/syn&ackBA)(ackAB/ackAB)(noBA/noBA)(ackAB/
    ackAB)
(synAB/synAB)(syn&ackBA/syn&ackBA)(finAB/-)(ackBA/-)
```

Figure 19: Example 2: Extract from the output file

Part IV. Conclusions

Finally a summary, some concluding notes to this work as well as some comments about further work to be done on this topic.

11. Summary

In this semester thesis, we implemented a tool for test case generation. This tool allows the user to draw a graphical representation of a Mealy Machine and then constructs input sequences with the expected output sequences.

We used JFLAP [Rod], a tool for interactive teaching of FSMs, as a foundation and extended it with the functionality to handle Mealy Machines. This graphical representation of Mealy Machines is then the starting point for the test case generation.

Test case generation can be done in several ways. In this semester thesis we implemented one way to do so. We focused on a general method which can be used in most cases. The used algorithms from [Gil62] for generating identification sets for each state work for every minimal FSM. The supervisor chose the Wp-method from [FvBK$^+$91] for generating test cases. This algorithm is applicable in more cases than similar algorithms.

The test case generation step is independent from the other parts of the tool. This modular design allows other test case generation algorithms to be added to the tool.

The implemented tool especially supports everything needed to test implementations of network protocols. The possibility to define the form and content of the used input and output symbols gives great flexibility without making the test case generation process more difficult since only the unique name of each symbol is used in the algorithms. The generated abstract test cases have then to be instantiated with the content defined.

Experiments with the tool showed that the used algorithms are general but not in every case optimal. The tool produces a few more test cases than needed for sufficient testing. This problem is inherent in the used algorithm and can not be easily solved.

12. Conclusions

Looking back at this semester thesis, I recognize the importance of a well chosen foundation and a complete theoretical background. Today, building up something from scratch is in most cases not necessary or affordable any more. But finding the right foundation is not that easy since there are so many possible candidates out there. I had already started to extend the Exorciser tool when I noticed that I only produced a mess in the source code. I then started again to look for another piece of software and came across JFLAP which now is the foundation for this tool.

When I started reading [FvBK+91], I did not expect many problems. The cover and identification sets were always presented as requirements and I found no word about the difficulty of generating them. I really hit the wall when I recognized that there is actually much more behind this test case generation than just these simple steps as described in [FvBK+91]. I crawled through several referenced papers and finally ended up with [Gil62], a book dating back in 1962. All papers about this test case generation methods just assumed that one knows the identification sets and in most cases did not even give a hint where to find out about that topic. It took me then about a week to get into that topic and to really understand the presentation in [Gil62], before I could go back to code and continue the actual work.

Enthusiastically about having mastered that problem, I overlooked that small paragraph in [Gil62] about the shortcomings of that method. When I finally had the code working that problem became apparent. But I was not able to find an alternative to this method and I had to accept that shortcoming.

But all in all I think this was an important experience for me. The work included everything from web research to reading and understanding quite difficult theoretical papers, from implementing algorithms to extending existing source code. Not to forget the writing of a report recalled the skill to express myself with words.

Sometimes I came with nothing to the weekly meetings with my supervisor. Sometimes I impressed her so much with my progress that she forgot to tell me some quite important things.

13. Future Work

This tool is a small part in a whole framework of tools to test firewalls. I solved the problem of abstract test case generation but it is just one step on the way down to a complete method to test firewalls.

The following list is not complete, but shows some tasks to improve this tool and points out the direction for getting further on the way to a method to test firewalls.

- *other test case generation algorithms:* The tool is designed to ease the extension with other test case generation algorithms. Different algorithms may solve some problems more efficiently than the chosen combination of methods.
- *implementing the algorithm more efficient:* Actually the algorithm for generating the test cases is implemented in Java, which probably is not the best choice from the performance viewpoint. From the XML-Output of the automaton, it is possible to attack the same problem with another implementation, such as with a functional language.
- *instantiating the abstract test cases:* As mentioned before, the test cases generated by this tool can not be used directly. The test cases have to be instantiated with additional information, e.g. testing an implementation of a TCP-protocol requires complete TCP-packets with IP-addresses, sequence numbers and so on, which is not provided by this tool.
- *allowing input and output alphabet to be different:* The actual implementation works with a single alphabet containing the input and output symbols of the automaton. This is no restriction for network protocols, since usually what can go in must also be allowed to go out and vice versa. But for other applications it is probably desirable to have distinctive alphabets for input and output.
- *adding other data types in the alphabet:* The offered data types cover the most basic data types used in computer science. Other data types can be added. Especially the notation of "'set" which is suggested in the GUI but not implemented could be an useful extension.
- *implementing special symbols:* I noticed that automata representing network protocols can become quite large. A few special symbols could improve readability of those automata: `>` for a packet that gets forwarded and `-` for a packet that gets dropped. With these symbols one could combine multiple transitions that indicate that the incoming packet gets forwarded by one single transition with multiple input symbols. The other special symbol is `other`, abstracting all input symbols that do not have a transition from that state. This makes drawing completely defined automata much easier.
- *more comfort in file handling:* Sometimes I really struggled with that file dialogue and the different file types the tool produced. A sophisticated file dialogue helping to distinguish all the files would be nice to have.

Part V.
Appendices

Additional documents to this thesis.

A. Task Description

Following the document describing the tasks for this semester thesis.

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Semester thesis for Stefan Hildenbrand

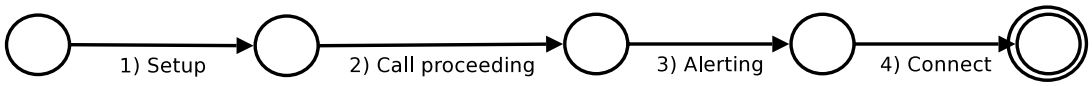
Generation of Test Cases

Programming of a tool for the generation of test cases from finite automata

Supervisor: Diana Senn
Professor: Prof. D. Basin
Issue Date: August 2005
Submission Date: February 2006

1 Introduction

We live in a world where all the company networks are connected to the Internet. Nobody can control the Internet, therefore a company has to protect their data from unauthorised access through the Internet. This is done by firewalls whose analogon in the physical world are locks. Everybody understands that doors need to be locked to prevent unauthorised access. It is the same in the digital world: unauthorised access to a company's network should be prevented, and this can be done by one or several firewalls.
Using the analogon of the door lock again, everybody understands that it is not enough to have a door lock. Only if the lock is locked properly and only authorised people have got a key to unlock it, we have what we want. It is the same in the digital world. It is not enough to have a firewall. We can only be satisfied if the firewall is doing what we expect from it. And to find out if a firewall satisfies our expectations (stated by a policy) we need to test it.



1) Setup

Source	A	
Destination	B	
Header	Field Name:	Protocol discriminator
	Field Value:	00001000
	Field Name:	Call reference
	Field Value:	X
	Field Name:	Message type
	Field Value:	00000101
	Field Name:	Sending complete
	Field Value:	[optional, Range = ?]
	...	
Payload	...	

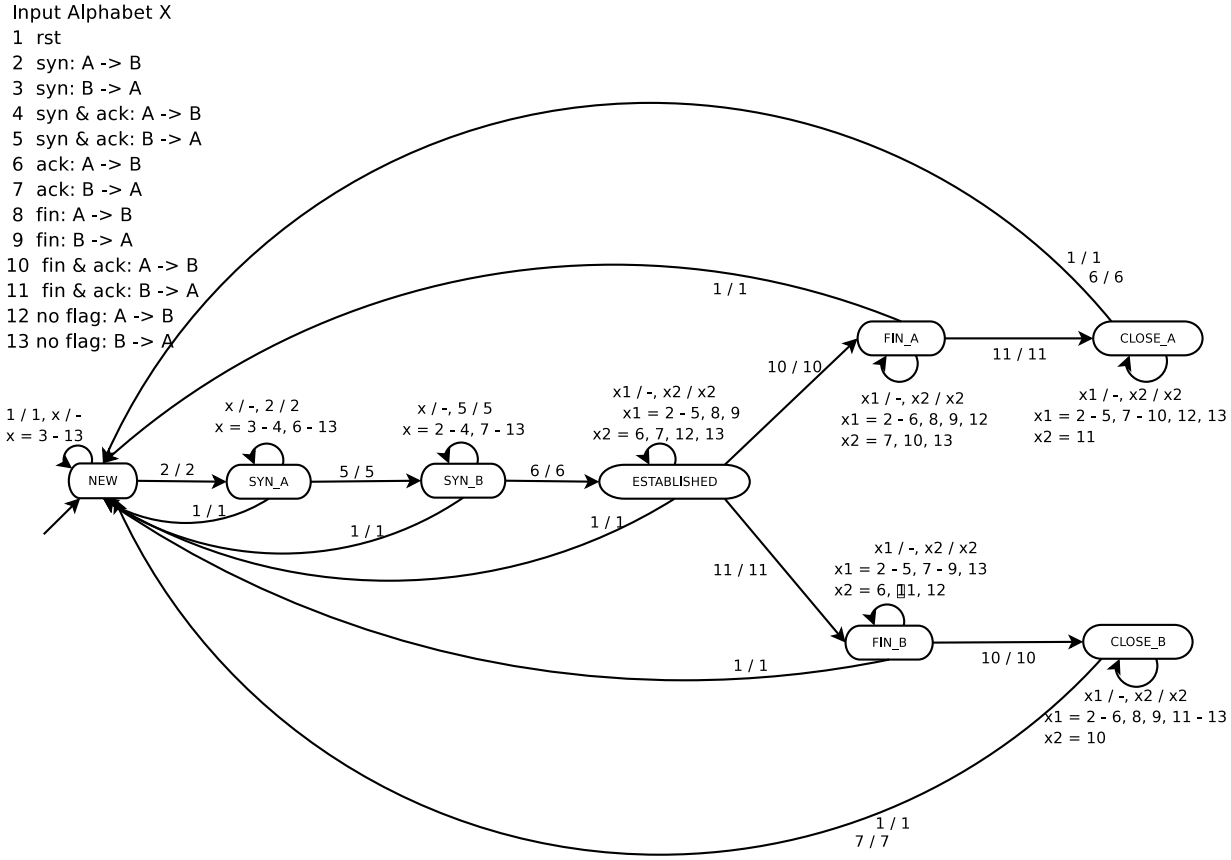
2) Call proceeding

Source	B	
Destination	A	
	Field Name:	Protocol discriminator
	Field Value:	00001000
	Field Name:	Call reference
	Field Value:	X
	Field Name:	Message type
	Field Value:	00000010
	Field Name:	Bearer capability
	Field Value:	[optional, Range = ?]
	...	
Payload	...	

Figure 1: Automaton for Simple Call Establishement in H.323

2 Motivation

When testing firewalls, one of many things that needs to be tested is the correct stateful handling of various protocols by the firewall. To do this, one needs a specification of the protocol which is to be tested. Such a specification can be written as a finite automata, see figures 1 and 2 for examples. From such automata, test cases can then be generated, using different methods [1, 2, 4, 5, 6, 7], and run against the firewall.



Some explanations:
I / O on transitions means Input and Output respectively.
The outputs are the reaction of a firewall to the given inputs. This is either accepting (forwarding) a packet or dropping it.
"x1 = 2-6" means $x1 \in 2..6$

Figure 2: Automaton for tcp

3 Assignement

3.1 Objectives

The goal of this project is to implement a tool which converts a graphical representation of an automaton into abstract test cases. These abstract test cases will then be instantiated with test tuples to generate concrete test cases [8], which then can be fed to fwtest [9].

3.2 Tasks

- Define criteria the tool has to satisfy (together with the supervisor)
- Evaluate tools for the graphical specification of automata, e.g. [3].
- Adapt the best suited tool to our needs
- Write a converter (for the tool chosen) between graphical and textual specifications of finite automata
- Generate abstract test cases from a textual representation of a finite automaton using an algorithm given by the supervisor

The whole software written during this thesis should rely on open source software (if possible) and should be modular and extendable. Particularly it should be easily possible to later on extend the software by other test generation algorithms.

3.3 Deliverables

- At the beginning of the semester thesis an agreement must be signed which allows the supervisor of this thesis, his project partners and ETH Zurich to use and distribute the software written during the thesis.
- At the end of the second week, a detailed time schedule of the semester thesis must be given and discussed with the supervisor.
- At the end of the semester thesis a presentation of 20 minutes must be given during an Infsec group seminar. It should give an overview as well as the most important details of the work.
- The final report may be written in English or German. It must contain an abstract written in both English and German, this assignement and the schedule. It should include an introduction, an analysis of related work, and a complete documentation of all used software tools. Three copies of the final report must be delivered to the supervisor.
- Software and configuration scripts developed during the thesis must be delivered to the supervisor on a CD-ROM.

16th August 2005

Prof. D. Basin

References

[1] Wendy Y. L. Chan, Son T. Vuong, and M. Robert Ito. An improved protocol test generation procedure based on UIOS. pages 283–294, 1989.

[2] Tsun S. Chow. Testing software design modeled by finite-state machines. In *IEEE Transactions on Software Engineering, Vol. SE-4, No 3*, pages 178–187, May 1978.

[3] Vincent Tscherter et al. Exorciser. http://www.educeth.ch/informatik/exorciser/.

[4] Susumu Fujiwara, Gregor von Bochmann, Ferhat Khendek, Mokhtar Amalou, and Abderrazak Ghedamsi. Test selection based on finite state models. *IEEE Trans. Softw. Eng.*, 17(6):591–603, 1991.

[5] A. Gill. State-identification experiments in finite automata. In *Information and Control, vol. 4*, pages 132 – 154, 1961.

[6] A. Gill. *Introduction to the Theory of Finite-state Machines*. McGraw-Hill, 1962.

[7] Krishan Sabnani and Anton Dahbura. A protocol test generation procedure. In *Computer Networks and ISDN Systems 15*, pages 285–297, 1988.

[8] Diana Senn, David Basin, and Germano Caronni. Firewall conformance testing. In Ferhat Khendek and Rachida Dssouli, editors, *Proceedings of TestCom 2005*, volume 3502 of *Lecture Notes in Computer Science*, pages 226–241. Springer-Verlag GmbH, May 2005.

[9] Gerry Zaugg. Firewall testing. http://www.infsec.ethz.ch/people/dsenn/DA_GerryZaugg_05.pdf.

B. Software Requirements Document

Following the document written at the beginning of this thesis fixing the requirements for the tool.

Software Requirements Specification (SRS)
Generation of Test Cases

24. August 2005

Inhaltsverzeichnis

1 Einführung

1.1 Zweck des Dokuments

Dieses Dokument soll das Software-Tool "Generation of Test Cases", das während der Semesterarbeit von Stefan Hildenbrand an der ETH Zürich entsteht, beschreiben. Es wendet sich sowohl an die Entwickler der Software als auch an die Benutzer, welche die Software einsetzen werden.

1.2 Zweck des Softwareprodukts

Das Tool soll eingesetzt werden, um eine graphische Repräsentation eines endlichen Automates in abstrakte Testfälle zu übersetzen. In der Welt der Informatik können viele Dinge als Endliche Automaten dargestellt werden. Um den korrekten Ablauf der Dinge sicherzustellen, müssen diese getestet werden. Dieses Tool soll zur Unterstützung solcher Tests dienen, indem es dem Benutzer erlaubt, den Vorgang (z.B. ein Kommunikationsprotokoll) als endlichen Automaten darzustellen und diese Darstellung in eine Folge von Input-Sequenzen mit den erwarteten Output-Sequenzen übersetzt. Diese Input-Sequenzen können dann der Implementation eingegeben werden und der reale Output mit dem erwarteten Output verglichen werden.

1.3 Erläuterungen zu Begriffen

Endlicher Automat Ein deterministischer Endlicher Automat (DFA, engl. deterministic finite automaton) ist ein Modell des Verhaltens, bestehend aus Zuständen, Zustandsübergängen und Aktionen. Ein Zustand speichert die Information über die Vergangenheit, d.h. er reflektiert die Änderungen der Eingabe seit dem Systemstart bis zum aktuellen Zeitpunkt. Ein Zustandsübergang zeigt eine Änderung des Zustandes des DFA und wird durch logische Bedingungen beschrieben, die erfüllt sein müssen, um den Übergang zu ermöglichen. Eine Aktion ist die Ausgabe des DFA, die in einer bestimmten Situation erfolgt.

Ein besonderer Typ DFA ist die *Mealy-Maschine*. Eine Mealy-Maschine wird formal als 7-Tupel beschrieben: $\mathcal{A} = (Q, \Sigma, \Omega, \delta, \lambda, q_0, F)$.

- Q ist eine endliche Menge von Zuständen ($|Q| < \infty$).
- Σ ist das Eingabealphabet. $|\Sigma| < \infty$
- Ω ist das Ausgabealphabet. $|\Omega| < \infty$
- δ ist die Übergangsfunktion $\delta : Q \times \Sigma \to Q$
- λ ist die Ausgabefunktion $\lambda : Q \times \Sigma \to \Omega$
- $q_0 \in Q$ ist der Startzustand.
- $F \subseteq Q$ ist eine (endliche) Menge möglicher akzeptierender Zustände. Wenn der Automat nach Lesen des Eingabewortes $w \in \Sigma^*$ in einem Zustand aus F hält, so gehört w zur Sprache $L(A)$.

In dieser Arbeit sind alle nicht näher definierten DFA Mealy-Maschinen.

Input- bzw. Output-Sequenz Eine Sequenz ist eine Folge von Eingaben bzw. Ausgaben welche der zu testenden Implementation als Eingabe zur Verfügung gestellt bzw als Ausgabe zu einer gegebenen Eingabe-Sequenz erzeugt wird.

Modell Das Modell bezeichnet die Anforderungen an die Implementation, die durch das Design oder die Policy vorgegeben werden.

zu testende Implementation Die zu testende Implementation (IUT - implementation under test) bezeichnet eine existierene Implementation des Modells, deren korrekte Arbeitsweise überprüft werden soll.

1.4 Übersicht über das restliche Dokument

Der Rest dieses Dokuments gibt eine detaillierte Beschreibung aller Requirements, welche das Tool erfüllen muss. Dabei geht es in Section 2 um die allgemeine Beschreibung der Software, dies beinhaltet einen generellen Produktüberblick, Überblick über die hauptsächlichen Funktionen der Software, Anforderungen an den Benutzer und Abhängigkeiten / Limitationen, die sich für die Software ergeben. Section 3 wird dann detailliert auf die spezifischen Anforderungen eingehen.

2 Beschreibung des Softwareprodukts

2.1 Zielsetzung und Einsatzumgebung des Produkts

Das Tool bildet ein Bindeglied zwischen dem Benutzer und weiteren Tools, welche in der Lage sind, die abstrakten Testfälle auf die entsprechende IUT anzuwenden. Das Ausgabeformat dieses Tools soll so generisch wie möglich gehalten werden, damit dieser Einsatzzweck erfüllt werden kann.

Für den Einsatz sollen aber möglichst geringe Anforderungen an bestehende Soft- oder Hardware gestellt werden. Eine Java-Umgebung darf vorausgesetzt werden.

2.2 Implementierte Funktionen

- der Benutzer kann beliebige Endliche Automaten erstellen, bestehend aus Zuständen und Übergängen
- der Benutzer kann für Ein- und Ausgabe jeweils ein beliebiges Format definieren, bestehend aus einer beliebigen Anzahl von Feldern und zugehörigem Datentyp
- für die Übergänge können entsprechend der Datentypen der Felder des Inputalphabets Einschränkungen definiert werden, z.B. die Zahl im Feld 4 muss zwischen 3 und 12 liegen
- die Einschränkungen für die einzelnen Felder können zu beliebigen Bool'schen Formeln verknüpft werden
- die definierten Einschränkungen können mit einem beliebigen Namen referenziert werden, der dann beim Übergang steht. Diese Namen dienen als Alphabet des DFA.

- das Format für Ein- und Ausgabe, sowie der DFA können in einem Format abgespeichert werden, das für weitere Tools einfach lesbar ist

2.3 Informationen zu erwarteten Nutzern

Zum effektiven Einsatz dieses Tools sind Kenntnisse im Bereich DFA und ein gutes Verständnis für die zu modellierenden Vorgänge notwendig. Für die Weiterverarbeitung der Ausgabe dieses Tools sind möglicherweise weitere Kenntnisse notwendig. Die Software richtet sich an erfahrene Software-Anwender, die mit Maus, Tastatur und einer Menü-gesteuerten Umgebung umgehen können.

2.4 Abhängigkeiten und Voraussetzungen

Eine Standard-Ausrüstung eines gängigen Desktop-Rechners sollte alle Anforderungen erfüllen. Eine Java-Umgebung wird vorausgesetzt.

3 Spezielle Anforderungen

3.1 Benutzeroberfläche (UI)

TG_UI_01 Das Tool bietet dem Benutzer eine graphische Oberfläche, wo er den DFA mit der Maus und den nötigen Werkzeugen (Zustand erstellen, Übergang erstellen) aufzeichnen kann.

TG_UI_02 Die Oberfläche bietet einen Button an, mit dem man zum Übergangsformat gelangt. Hier kann der Benutzer in Form von zwei Tabellen das Format für Ein- und Ausgabe definieren (Name der einzelnen Felder eingeben sowie Datentyp wählen).

3.2 Schnittstellen (IF)

TG_IF_01 Das Format für die Übergänge und die gezeichneten DFA können abgespeichert werden und ebenfalls wieder eingelesen werden.

TG_IF_02 Das Tool bietet eine Schnittstelle in Form einer strukturieren Datei an, so dass weitere Tools den gezeichneten DFA verarbeiten können.

TG_IF_03 Das Tool erzeugt eine für *fwtest* geeignete Ausgabe von Test Cases.

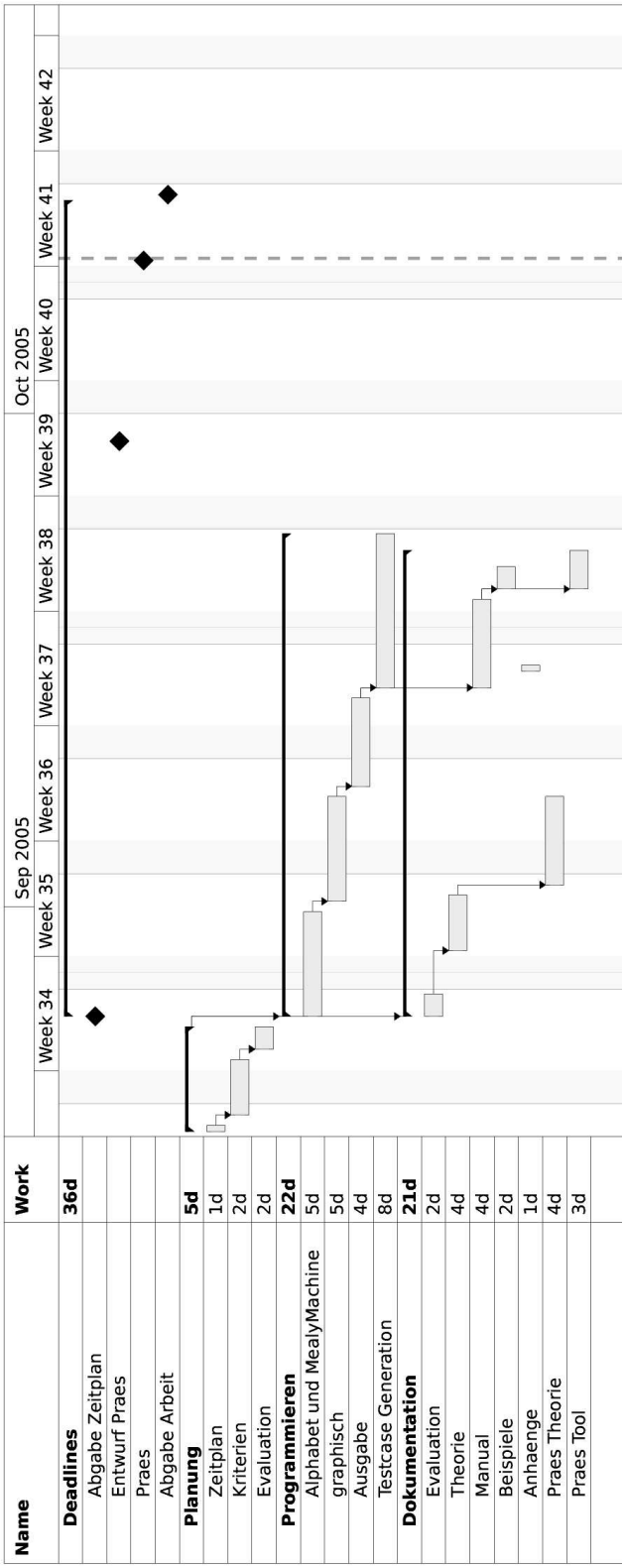
3.3 Funktionalität (FUN)

TG_FUN_01 Die gezeichneten DFA können auf Anweisung des Benutzers automatisch angeordnet werden.

TG_FUN_02 Das Tool enthält einen Algorithmus zur Generierung von Test Cases und erzeugt auf Anweisung des Benutzers aus dem aktuellen DFA eine entsprechende Ausgabe.

C. Schedule

Following the schedule used for the work at this project.



D. CD contents

Following the description of the CD contents that also appears in the root directory of the CD.

```
*******************************************************
*                        TCGTOOL                      *
*           automated test case generation            *
*             from finite state machines              *
*******************************************************

CD CONTENTS
===========

on this CD-ROM the following content can be found:

/
   this file

/doc
   The report of this semester thesis, the presentation slides
   used in an intern talk and additional documentation.

/doc/src
   The tex source files for the documentation and the slides.
   Call "make" in this directory to compile the documentation.

/src
   The Java source files for TCGTool.
   Call "make" in this directory to compile the tool.

/bin
   A jar package of TCGTool, already compiled, just start it
   using your Java Runtime Environment.

/opt
   Additional files and packages used for TCGTool.
   /opt/jflap -  the tool used as foundation for TCGTool
   /opt/java  -  the Java 1.4.2 JRE for Windows and Linux/x86
   /opt/latex -  some additional packages and files necessary
                 to compile the documentation.
```

E. Readme

Following the Readme coming with the tool.

```
*****************************************************
*                       TCGTOOL                     *
*           automated test case generation          *
*             from finite state machines            *
*****************************************************

This tool was developed as part of a semester thesis at
the Swiss Federal Institute of Technology.

The report of this thesis can be found at:

http://www.infsec.ethz.ch/people/dsenn/SA_StefanHildenbrand_05.pdf

The tool comes with an online help. All information can
be found in the report and the online help.

Building TCGTool
----------------

Compiling and running TCGTool requires Java 1.4.  In order to
build TCGTool and the JAR, a Makefile has been provided.
To build TCGTool with gnumake, enter

        make

from the command line.  This will create the compiled '.class'
Java files, as well as the TCGTool.jar executable jar file.
The script used to create the TCGTool.jar executable requires
Python 2.2.

To run from the class files:

        java TCGTool

To run as a JAR:

        java -jar TCGTool.jar

Alternatively on OS X and on a correctly configured Windows
machine, you can double click the TCGTool.jar file and it
should just work.
```

References

[Cho78] Tsun S. Chow. *Testing Software Design Modeled by Finite-State Machines*. In *IEEE Transactions on Software Engineering*, volume 4, pages 178–187, May 1978.

[ea] Vincent Tscherter et al. *Exorciser: Automatic Generation and Interactive Grading of Exercises in the Theory of Computation*. `http://www.swisseduc.ch/exorciser`.

[FvBK+91] Susumu Fujiwara, Gregor von Bochmann, Ferhat Khendek, Mokhtar Amalou, and Abderrazak Ghedamsi. *Test Selection Based on Finite State Models*. In *IEEE Transactions on Software Engineering*, volume 17, pages 591–603, June 1991.

[Gil62] A. Gill. *Introduction to the Theory of Finite-state Machines*. McGraw-Hill, 1962.

[HL01] Christian Häfeli and Reto Lamprecht. *Interactive Learning Components for the Study of Finite Automata*. ETH Zürich, March 2001.

[Rod] Susan H. Rodger. *JFLAP - Java Formal Language and Automata Package*. `http://www.jflap.org`.

[var] various. *Wikipedia*. `http://www.wikipedia.org`. The Free Encyclopedia.

[Zau] Gerry Zaugg. *Firewall Testing*. `http://www.infsec.ethz.ch/people/dsenn/DA_GerryZaugg_05.pdf`.