# JavaFSM

Studienarbeit

von Karola Krönert und Ulrich Dallmann
Universität Hamburg
Fachbereich Informatik
Arbeitsbereich TECH

1 Einführung ..... 3
1.1 Einleitung ..... 3
1.2 Aufgabenstellung ..... 3
1.3 Gliederung der Arbeit ..... 3
2 Was ist eine FSM? ..... 4
3 Aufbau eines Schaltwerkes ..... 5
4 Java und Objektorientierung ..... 7
4.1 Entwicklungsumgebungen ..... 8
5 Implementation und Programmaufbau ..... 8
5.1 FSM ..... 9
5.2 "Just-in-time-Parsing" ..... 10
5.3 Methode zur Überprüfung eines eingegebenen Automaten ..... 10
5.4 Selektionsalgorithmus im Editor ..... 10
5.5 Laden und Speichern ..... 11
5.5.1 Laden und Speichern aus dem Applet mit CGI (Common Gateway Interface) ..... 12
5.6 Parser ..... 13
5.7 Export von Automaten nach VHDL / KISS ..... 13
5.7.1 VHDL ..... 13
5.7.2 KISS / PLA Design Format ..... 14
5.8 Beschreibung der Klassen ..... 15
6 Bedienungs-Anleitung ..... 16
6.1 JavaFSM als Applikation ..... 16
6.2 Auswahl zwischen Mealy- und Moore-Schaltwerken ..... 17
6.3 Speichern und Laden von entworfenen Automaten ..... 17
6.3.1 Firewall / Proxy ..... 17
6.4 Laden von Beispielautomaten ..... 17
6.5 Festlegen des Automaten-Namen ..... 18
6.6 Festlegen der Ein-/ und Ausgänge ..... 18
6.7 Entwurf des zugrundeliegenden Automaten mit Hilfe des Editors ..... 18
6.8 Definition der logischen Übergangsbedingungen und Ausgabefunktionen und Ändern der Zustandsnamen ..... 19
6.9 Überprüfung des Automaten auf Korrektheit ..... 19
6.10 Zwischen Editier- und Simulationsmodus hin- und herschalten ..... 19
6.11 Simulation durch Verändern der Eingangswerte und Taktgebung ..... 19
6.12 Ausgabe des Simulationsergebnisses im Impulsdiagramm ..... 19
6.13 Export des Automaten im VHDL/ KISS-Format ..... 19
7 Beispiel: Counter ..... 20
8 Ein komplexeres Beispiel: Fahrstuhl ..... 25
9 Kritische Betrachtung ..... 26
10 Zusammenfassung und Ausblick ..... 27
11 Literaturverzeichnis ..... 28
12 Anhang A ..... 29
13 Anhang B ..... 31
14 Anhang C: package JavaFSM ..... 33

# 1 Einführung

## 1.1 Einleitung

Durch die steigende Komplexität und Integration im Chipdesign werden immer komplexere Ablaufsteuerungen erforderlich. Daher gewinnen endliche Automaten zur Modellierung in diesem Gebiet zunehmend an Bedeutung. Trotzdem fehlt es hier noch an geeigneten grafischen Entwicklungstools, insbesondere im Bereich Public Domain Software.
Außerdem entstand im Arbeitsbereich TECH der Universität Hamburg der Wunsch, den Studenten im Grundstudium die Grundlagen der Technischen Informatik durch praktische Beispiele näher zu bringen. So entstand die Idee ein Programm zu entwickeln, das beiden Anforderungen gerecht wird. JavaFSM ist also sowohl für Studenten zur Einführung, als auch für Entwickler zum praktischen Einsatz gedacht.

## 1.2 Aufgabenstellung

Im Rahmen dieser Arbeit soll ein interaktiver Simulator für Mealy- und Moore-Schaltwerke erstellt werden. Dabei sollen zunächst die Ein- und Ausgänge des Schaltwerkes festgelegt werden können, wobei die Wahl zwischen dem Mealy- und dem Moore-Modell besteht. In einem Editor wird dann der Automat mit Zuständen, Transitionen und Übergangsbedingungen entworfen und ein Startzustand festgelegt. Danach kann der Automat simuliert und das Ergebnis in einem Impulsdiagramm ausgegeben werden. Ferner ist es möglich, entworfene Automaten zu speichern, zu laden und nach VHDL oder KISS zu exportieren.

Aufgrund der besonderen Eigenschaften ist als Programmiersprache für den Simulator Java vorgesehen (siehe hierzu insbesondere Kapitel 4).
Um als Hilfsmittel bei der Einführung von Studenten zu dienen, sollte das Programm intuitiv bedienbar sein und über eine graphische Benutzeroberfläche verfügen. Zusätzlich sollte es möglichst einfach zugänglich sein. Um auch den Entwickler von Schaltwerken zu unterstützen, muß das Programm ausreichend leistungsfähig sein.

## 1.3 Gliederung der Arbeit

In Anschluß an dieses einführende Kapitel wird im nächsten Kapitel der endliche Automat als formales Modell der Informatik vorgestellt. Kapitel 3 gibt eine Einführung in den Aufbau und die Funktionsweise von Schaltwerken.
Als Programmmiersprache für JavaFSM wurde die neue, objektorientierte Sprache Java von Sun Microsystems gewählt. Java und dessen Besonderheiten sind Inhalt von Kapitel 4.
In Kapitel 5 werden einige Implementations-Details und der grundsätzliche Programmaufbau beschrieben.

Eine Bedienungsanleitung von JavaFSM findet sich in Kapitel 6. In Kapitel 7 wird die Verwendung von JavaFSM anhand eines einfachen Beispiels erläutert. Kapitel 8 stellt dann ein etwas komplexeres Beispiel vor.
In Kapitel 9 findet eine Bewertung des Erreichten statt und in Kapitel 10 wird eine Zusammenfassung und einen Ausblick auf mögliche Erweiterungen gegeben.
In Anhang A und B finden sich die beiden CGI-Scripte, die zum Laden und Speichern von Automaten verwendet werden. Anhang C enthält die Übersichtsseite der Javadoc Klassen-Dokumentation.

# 2 Was ist eine FSM?

Eine FSM (FiniteStateMachine, endlicher Automat) ist ein formales Modell, durch das sich viele Probleme der Informatik beschreiben lassen. In diesem Programm werden endliche Automaten zur Modellierung von Schaltwerken verwendet.
Ein Automat beschreibt ein System, das sich in einer aus einer endlichen Anzahl von internen Konfigurationen- sogenannten Zuständen befindet. Der Zustand des Systems umfaßt die Informationen, die sich aus den bisherigen Eingaben ergeben haben und die benötigt werden, um die Reaktion des Systems auf die noch folgenden Eingaben zu bestimmen [HU93]. Einer dieser Zustände wird als Startzustand definiert. Das System kann nun unter bestimmten Bedingungen den Zustand wechseln. So einen Zustandsübergang nennt man im Automatenmodell Transition. Graphisch stellt man Zustände durch Kreise und Transitionen durch Pfeile - beschriftet mit den Übergangsbedingungen - dar (siehe Abbildung).
Formal besteht ein Automat (A) aus einer endlichen Menge von Zuständen (Q), einem Eingabealphabet (I), einer Übergangsfunktion (d), einem Startzustand (q0) und einer Menge von Endzuständen (F).

$$A = \left(Q, I, d, q_0, F\right)$$

Bei der Modellierung von Schaltnetzen entfällt die Menge der Endzuständen, da es hier nicht um das Akzeptieren von Wörtern, sondern um die Berechnung von Ausgabewerten und Zustandsübergängen geht. Das Eingabealphabet ist $(0,1)^n$, wobei n die Anzahl der Eingänge ist.

**Beispiel:**

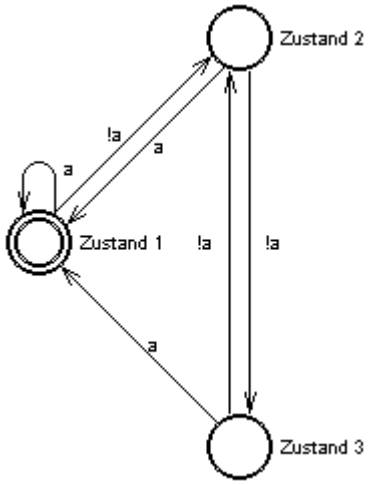
Q={Zustand 1, Zustand 2, Zustand 3}

q0=Zustand 1

I=(0,1) (Werte für a)

d(q,z) → (q')

z.B.: d(Zustand 1, !a) → (Zustand_2)



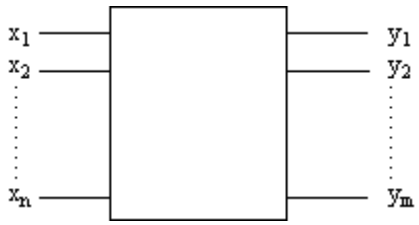

Grundsätzlich gibt es deterministische und nicht-deterministische Automaten. Bei deterministischen Automaten sind die Übergänge immer eindeutig definiert, während es bei nicht-deterministischen mehrere Übergangsmöglichkeiten geben kann, von denen der Automat die richtige "raten" muß. Zum Entwurf von Schaltwerken werden nur deterministische Automaten verwendet, weshalb sie auch diesem Programm zugrunde liegen.
In JavaFSM wird nicht, wie im formalen Modell, eine einzige Übergangsfunktion angegeben. Stattdessen wird für jede Transition eine Übergangsbedingung festgelegt. Da es sich, wie oben erwähnt, um einen deterministischen Automaten handelt, ist jeweils genau eine der Transitionen aktiviert.
Im Chipdesign kommen endliche Automaten beim Design von getakteten Schaltwerken zum Einsatz, da sie die Umsetzung vieler elektronischer Funktionen, insbesondere

Ablaufsteuerungen, stark vereinfachen. Um beim Entwurf unhandliche und fehleranfällige Logiktabellen zu vermeiden, bieten neuere Hardware-Beschreibungs-Sprachen deshalb oft direkte Syntax für endliche Automaten an.

# 3 Aufbau eines Schaltwerkes

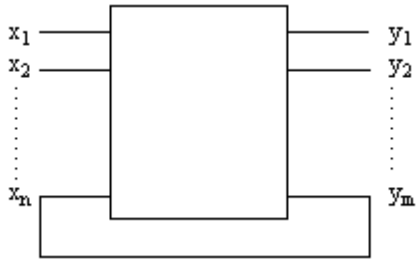
Dieses Kapitel gibt eine Einführung in den Aufbau und die Funktionsweise von Schaltwerken.

Die Verarbeitung von binären Informationen erfolgt in digitalen Rechenanlagen mit Hilfe logischer Schaltungen. Ein Schaltnetz hat n Eingangs- und m Ausgangssignale:




**Schaltnetz**

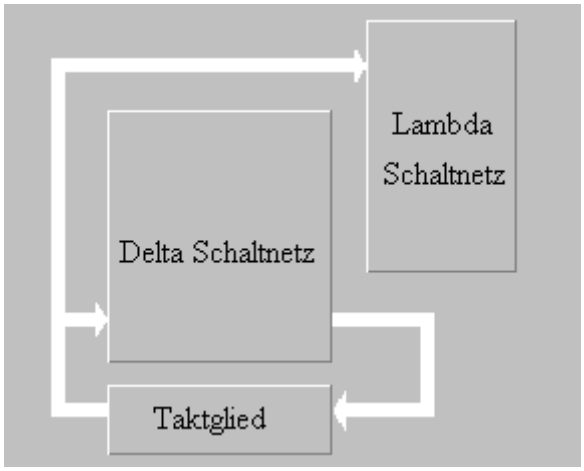
An den Eingängen können binäre Werte x1,x2,..., xn mit xi aus {0,1}eingegeben werden. An den Ausgängen liegen in Abhängigkeit von den Eingabesignalen binäre Worte y1,y2,..., ym mit yi aus {0,1}an.
Hängt bei einer logischen Schaltung die Ausgabe nur von der momentan anliegenenden Eingabe ab, so nennt man diese **Schaltnetz**. Hängt die Ausgabe außerdem noch von früheren Eingaben ab (Rückführung der Ausgänge), so handelt es sich um ein **Schaltwerk**. Mittels der Rückführung lassen sich Informationen speichern.



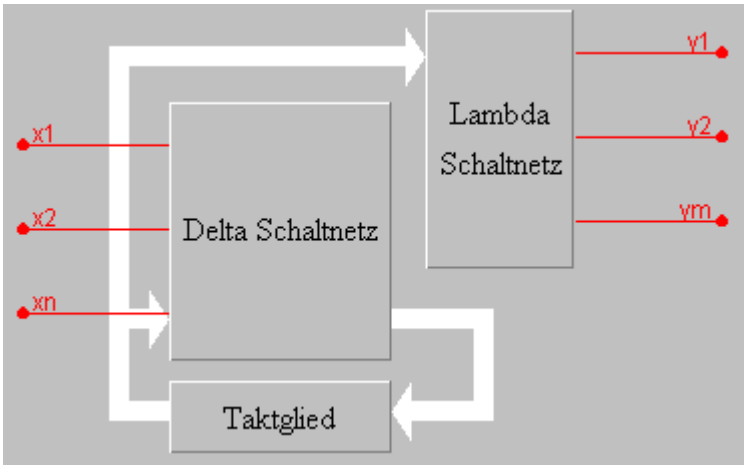

**Schaltwerk**

Da bereits bei kleinen Schaltwerken direkte Rückführungen äußerst schwer handhabbar sind (Hazards), fügt man in die Rückführung ein Taktglied ein. Zur besseren Übersicht teilt man das Schaltnetz in zwei Teile: der erste Teil dient zur Bestimmung der Folgezustände (Delta-Schaltnetz), während der zweite die Ausgabewerte berechnet (Lambda-Schaltnetz) [La87].

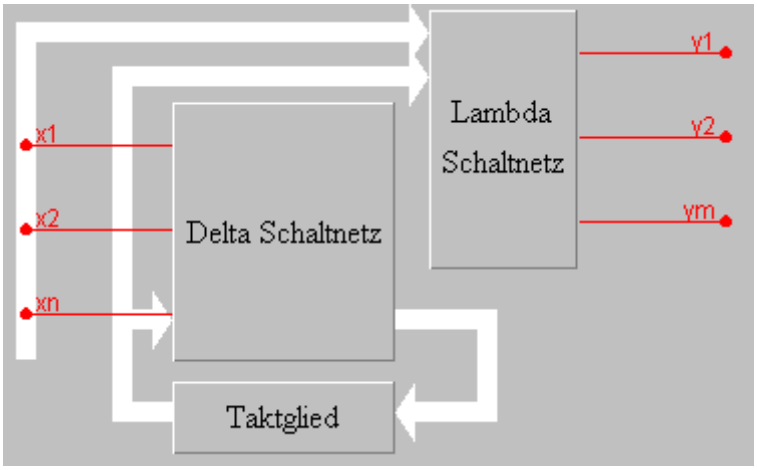
**Das Schaltwerk ist damit wie folgt aufgebaut:**




- **Delta-Schaltnetz:** hier wird aus dem Zustand, in dem sich der Automat gerade befindet, und der aktuellen Eingangsbelegung der Nachfolgezustand berechnet.
- **Taktglied:** hier wird mittels Flip-Flops gespeichert, in welchem Zustand sich der Automat gerade befindet. Bei einem Taktsignal wird der gerade vom Delta-Schaltnetz berechnete Folgezustand als neuer Zustand übernommen und dem Delta- sowie Lambda-Schaltnetz bereitgestellt. Eine Zustandsänderung kann also nur bei einem Taktsignal erfolgen.
- **Lambda-Schaltnetz:** hier werden die aktuellen Ausgangsbelegungen berechnet. Bei einem **Moore**-Automaten hängen diese nur vom aktuellen Zustand ab. Bei **Mealy**-Automaten hingegen werden auch die Eingänge in die Berechnungen einbezogen. Hierdurch können auch zwischen den Zustandsübergängen Änderungen der Ausgangsbelegung erzeugt werden. Letztlich sind Mealy-Automaten jedoch nicht mächtiger, da sich zeigen läßt, daß es (abgesehen von einem Zeitversatz) immer einen äquivalenten Moore-Automaten gibt.




**Moore-Schaltwerk**




**Mealy-Schaltwerk**

Ein wichtiges Beschreibungsmittel für Schaltwerke sind endliche Automaten. JavaFSM verwendet diese daher zum Entwurf und bietet zudem die Möglichkeit, erstellte Automaten als Schaltwerke nach VHDL und KISS zu exportieren.

# 4 Java und Objektorientierung

In diesem Kapitel werden die generellen Eigenschaften und Besonderheiten der Programmiersprache Java vorgestellt. Außerdem soll gezeigt werden, warum Java sich besonders für dieses Projekt eignet.

Java bietet gegenüber herkömmlichen Programmiersprachen mehrere Vorteile:

Java ist plattformunabhängig. Ein Java-Programm wird von einem Compiler in plattfomunabhängigen Bytecode übersetzt, der dann zur Laufzeit von einem Interpreter, der Java Virtual Machine, ausgeführt wird. Die Virtual Machine ist mittlerweile für alle gängigen Systeme implementiert.

Java ist objektorientiert. Bei der Objektorientierung stehen die Daten und deren Manipulation im Vordergrund, nicht die Prozeduren. Objekte können ihre Eigenschaften an andere Objekte vererben oder um weitere Eigenschaften ergänzt werden.

Java bietet umfangreiche Klassen-bibliotheken, die neben Standard-Funktionen auch neue Bereiche, wie Netzwerk- und Datenbankzugriff abdecken. Da sich die Bibliotheken bereits beim Client im Java-Interpreter befinden, müssen diese nicht über das Netz geladen werden.

Neben Standalone-Applikationen bietet Java die Möglichkeit, Programme in HTML-Seiten einzubetten. Diese sogenannten Applets werden dann von der in den Browser integrierten Virtual Machine ausgeführt. Java-Applets sind daher über das World-Wide-Web weltweit zugänglich. Ein weiterer Vorteil von Applets ist, daß keine Installation notwendig ist. Auch Updates passieren durch einen einfachen Austausch der Klassen auf dem WWW-Server.

Durch die Plattformunabhängigkeit von Java und die Möglichkeit Applets in HTML-Seiten einbetten zu können, eignet sich Java besonders gut für den Einsatz in der Lehre, da hierdurch viele Leute erreicht werden können.

Referenzplattform für Java ist das Java Developers Kit (JDK) der Firma Sun.

Das JDK bietet mit dem Befehl javadoc die Möglichkeit an, aus dem Sourcecode und den Kommentaren automatisch eine Dokumentation der Klassen in HTML zu generieren.

### 4.1 Entwicklungsumgebungen

JavaFSM entstand auf Basis des JDK (Java Developers Kit) Version 1.0.2 der Firma Sun Microsystems, welches kommandozeilen-orientiert arbeitet und daher zur Programmentwicklung nicht sehr komfortabel ist.

Zu Beginn dieser Arbeit waren nur wenige Entwicklungstools verfügbar. Daher haben wir für die Entwicklung zunächst einen erweiterten Editor benutzt, der die Syntax der Sprache farblich hervorhebt und das Compilieren per Mausklick ermöglicht (Javamaker und Jpad).

Nach und nach erschienen jedoch von mehreren Herstellern integrierte Entwicklungsumgebungen. Wir haben uns dabei für Microsoft Visual J++ entschieden. Allerdings gibt es hier gewisse Kompatibilitäts-Probleme, weshalb es ratsam ist, abschließend das Programm mit dem JDK zu compilieren.

Als Literatur zur Programmiersprache Java dienten uns [LP96], [CH96], [Fla96], [AG96] und [He97].

# 5 Implementation und Programmaufbau

Wie im vorigen Kapitel erläutert, wurde JavaFSM als Java Programm realisiert. Dabei läßt es sich sowohl als Applet, als auch als Applikation starten.
Grundsätzlich besteht JavaFSM aus drei Fenstern: in dem ersten ist das Schaltwerkmodell mit den Ein- und Ausgängen zu sehen. Hier kann ein Automaten-Name vergeben und der Schaltwerk-Typ festgelegt werden. Außerdem können die Ein- und Ausgänge hinzugefügt, geändert und gelöscht werden. Nach Eingabe des Automaten kann dieser hier simuliert werden. Dazu gibt es einen Takt- und einen Reset-Button. Im Delta-Schaltnetz ist der Automat sichtbar. Der aktuelle Zustand und die unter der momentanen Belegung aktivierte Transition sind rot markiert.

Im Editorfenster wird der Automat entworfen. Hier werden Zustände, Transitionen, sowie Übergangs- und Ausgabefunktionen eingegeben. Zusätzlich besteht die Möglichkeit einen Automaten mit Kommentaren zu versehen. Der Editor arbeitet Modus-orientiert. Über Buttons kann man jeweils den Modus wechseln. Es gibt 6 Modi: im Bewegen-Modus lassen sich Zustände verschieben. Außerdem können Transitionen und Zustände selektiert werden, um im Parameterfeld deren Eigenschaften (Name, Funktion, etc.) zu verändern (siehe auch

Selektionsalgorithmus). Je nachdem, ob ein Moore- oder Mealy-Automat editiert wird, ändert sich das Parameterfeld der Zustände. Während jeder Output bei Moore feste Ausgabewerte hat, kann bei Mealy jeweils eine Funktion (über den Inputs) eingegeben werden. Auf diese Weise läßt sich das Lambda-Schaltnetz "simulieren", ohne das eine konkrete Zustandskodierung existiert.

Der Zustand-Modus fügt per Mausklick Zustände ein. Im Transition-Modus werden zwei Zustände jeweils durch Anklicken mit einer Transition verbunden. Die Transitionen werden als Pfeile zwischen zwei Zuständen dargestellt. Der Algorithmus zum Zeichnen der Pfeile wurde dem Programm JavaFIG in [He97] entnommen. Durch Anklicken im Lösch-Modus können Komponenten wieder entfernt werden. Den Startzustand legt man im Startzustand-Modus fest. Im Kommentar-Modus lassen sich auf der Zeichenfläche Kommentare einfügen. Da während der Simulation der Automat nicht verändert werden darf, wird das Editorfenster dabei automatisch geschlossen.

Das dritte Fenster mit dem Impulsdiagramm bietet die Möglichkeit, sich die Simulationsergebnisse darstellen zu lassen. Dabei werden die folgenden Daten benötigt: die Zustandsfolge (im Vektor "zustandsfolge" der Klasse FSM), die In- und Outputs (in den Vektoren "inputs" und "outputs" der Klasse FSM) und die Wertefolgen der In- und Outputs (im Vektor "taktfolge" der Klasse Signal für jeden In- und Output). Zu besseren Unterscheidung werden die Inputs blau und die Outputs rot in verschiedenen Farbabstufungen dargestellt.

Ein Problem ergab sich im Impulsdiagramm bei Mealy-Automaten. Hier können sich die Ausgangswerte auch zwischen den Takten ändern, wenn sich die Eingangswerte ändern. Hier haben wir sogenannte Zwischentakte eingeführt. Jede Änderung der Eingangswerte generiert so einen Zwischentakt. Zusätzlich wurde ein Vektor "taktfolge" in die Klasse FSM eingefügt, der für jeden echten Takt ein "true" enthält und für jeden Zwischentakt ein "false". Im Impulsdiagramm werden nun bei Mealy-Automaten sowohl echte als auch Zwischentakte angezeigt. Taktlinien und Beschriftung werden jedoch nur bei echten Takten eingefügt.

## 5.1 FSM

Zur Repräsentation und Verarbeitung endlicher Automaten wurde die Klasse FSM entworfen. Hier werden alle Daten des Automaten zusammengefaßt. Außerdem stellt die Klasse alle Grundfunktionen zum Bearbeiten zur Verfügung.
Die Basiskomponenten eines endlichen Automaten sind Zustände, Transitionen, Inputs und Outputs. Deren Eigenschaften sind in der folgenden Tabelle aufgelistet:

| Zustand | Transition | Input | Output |
|---|---|---|---|
| • Name<br>• Position<br>• Ausgabefunktion<br>• Startzustand | • Ausgangszustand<br>• Zielzustand<br>• Übergangs-bedingung | • Name<br>• Initialwert<br>• aktualler Wert<br>• Wertefolge der Simulation | • Name<br>• aktueller Wert<br>• Wertefolge der Simulation |

Die dynamischen Werte der Simulation werden jeweils direkt bei dem entsprechenden Objekt gespeichert, d.h.

- jeder In- und Output enthält einen Vektor mit der Werte-Folge seiner Belegung
- die Klasse FSM enthält einen Vektor mit der Zustandsfolge
- da sich beim Mealy-Automaten die Ausgänge nicht nur nach Takten ändern können, enthält FSM zusätzlich einen Vektor "taktfolge", der eine Unterscheidung von Takten und „Zwischentakten“ ermöglicht

## 5.2 "Just-in-time-Parsing"

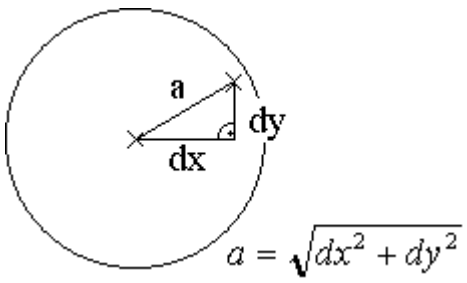
Die Berechnung von Übergangstabellen für den endlichen Automaten erfordert exponentiellen Rechenaufwand und zusätzlichen Speicher. Bei einer Änderung des Automaten müssen die Tabellen spätestens zur Simulation neu berechnet werden. Die in JavaFSM verwendete Alternative ist das "Just-in-time"-Parsen der jeweils benötigten Funktionen während der Simulation. Dabei werden nur Transitionen betrachtet, die vom aktuellen Zustand ausgehen.
Zur Berechnung der Übergangsbedingungen und der Ausgabefunktionen in den Zuständen dient die Klasse Parser, die die jeweilige Funktion als String übergeben bekommt.
Größerer Rechenaufwand (exponentiell) fällt nur beim Überprüfen von Automaten („FSM testen“-Button im Editor) und beim Export nach KISS an, da hier sämtliche Funktionen mit allen möglichen Belegungen berechnet werden müssen.

## 5.3 Methode zur Überprüfung eines eingegebenen Automaten

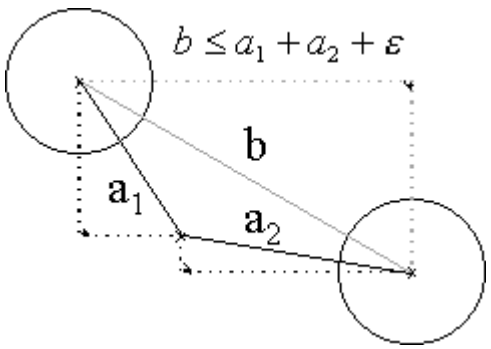
Der Editor enthält eine Funktion, die es erlaubt, den Automaten auf syntaktische Fehler zu überprüfen.
Der "FSM-Tester" überprüft zunächst, ob überhaupt Zustände und Transitionen vorhanden sind, und ob einer der Zustände als Startzustand definiert wurde. Danach geht er alle möglichen Belegungen durch und stellt fest, ob jeweils genau eine Transition aktiviert ist. In einem Dialogfenster werden dabei Statusmeldungen und gegebenenfalls Fehler ausgegeben.

## 5.4 Selektionsalgorithmus im Editor

Die Selektion von Zuständen erfolgt einfach durch die Berechnung der Entfernung zwischen Zustandsmittelpunkt und Klickpunkt (siehe Abbildung). Um Transitionen zu selektieren, müßte eigentlich die Entfernung vom Klickpunkt zur Linie (Lot) berechnet werden. Eine einfachere Möglichkeit besteht im Vergleich der Entfernung beider Zustände mit der Summe der Entfernungen der Zustände zum Klickpunkt. Ist diese nur unwesentlich größer, dann wurde auf bzw. direkt neben die Transition geklickt:




Zustandsselektion




Transitionsselektion

## 5.5 Laden und Speichern

JavaFSM bietet die Möglichkeit, selbst entworfene Automaten zu speichern und wieder zu laden.
In der Applikation wird dabei über die Menüpunkte Laden und Speichern ein File-Requester geöffnet, über den sich eine Datei auswählen läßt.
Im Applet wird in einem Dialogfenster ein Name und ein Paßwort angegeben und der Automat unter diesem Namen auf dem Server abgelegt.
**Eine FSM-Datei hat den folgenden Aufbau:**

| | |
|---|---|
| [JavaFSM V1.0] | Header mit Versionsnummer |
| [Machine Type] | MEALY oder MOORE |
| [Name] | Kennzeichnet den Namens-Block |
| Name | gibt den Namen des Automaten an |
| [Inputs Anzahl] | Kennzeichnet den Input-Block, wobei Anzahl angibt wieviele Inputs es gibt |
| name,initial | Name und Startwert eines Inputs (jede Zeile enthält genau einen Input) |
| [Outputs Anzahl] | Kennzeichnet den Output-Block, wobei Anzahl angibt wieviele Outputs es gibt |
| name | Name eines Outputs (jede Zeile enthält genau einen Output) |
| [Zustaende Anzahl] | Kennzeichnet den Zustands-Block, wobei Anzahl angibt wieviele Zustände es gibt |
| name,xpos,ypos,isStart,output1_fkt... | Name, Position, isStart (boolean) und für jeden Output die Ausgabefunktion |
| [Transitionen Anzahl] | Kennzeichnet den Transitions-Block, wobei Anzahl angibt wieviele Transitionen es gibt |
| name1,name2,fkt | Namen der beiden Zustände und die Übergangsfunktion |
| [Kommentare Anzahl] | Kennzeichnet den Kommentar-Block, wobei Anzahl angibt wieviele Kommentare es gibt |
| xpos, ypos, Text | Position und Kommentar-Text (dieser kann "%10" für einen Zeilenumbruch enthalten) |
| [ENDE] | Markiert das Ende |

**Beispiel:**
[JavaFSM V1.0]
[MOORE]
[Name]
Beispiel-Automat
[Inputs 2]
a,0
b,1
[Outputs 3]
x
y
z
[Zustaende 2]
Zustand 1,58,129,true,0,1,0

Zustand 2,229,222,false,1,0,1
[Transitionen 4]
Zustand 2,Zustand 1,b
Zustand 1,Zustand 1,*
Zustand 1,Zustand 2,a
Zustand 2,Zustand 2,*
[Kommentare 1]
10,10,Dies ist ein Beispielautomat
[ENDE]

### 5.5.1 Laden und Speichern aus dem Applet mit CGI (Common Gateway Interface)

Ein Java Applet kann aus Sicherheitsgründen nicht auf das Filesystem des Rechners zugreifen. Eine Möglichkeit zur Speicherung von Automaten besteht darin, Text in einem Fenster auszugeben und diesen mittels Copy&Paste zu speichern. Diesen Weg gehen wir bei der Konvertierung des Automaten nach VHDL / KISS, damit der Anwender die Daten direkt weiterverarbeiten kann.
Das Versenden einer e-mail mit den Daten an den User stellt eine weitere Möglichkeit zur Datensicherung dar. Diese löst jedoch noch nicht das Problem des Ladens von Automaten. Die Lösung, die wir zum Laden und Speichern von Automaten gewählt haben, liegt in einer Netzwerkverbindung zum Hostrechner. Statt einen eigenen Server in Java zu implementieren, bot sich die Verwendung des vorhandenen WWW-Servers an. Ein WWW-Server bietet mit dem CGI (Commin Gateway Interface) die Möglichkeit, Programme zu starten [He96]. Diese CGI-Programme werden über eine bestimmte URL aufgerufen. Sie können beispielsweise dynamisch HTML-Seiten generieren (z.B. Suchmaschinen).
Die Kommunkation zwischen Browser, Server und CGI-Programm läuft folgendermaßen ab: der Browser sendet eine Anfrage an den Server und erwartet darauf eine Antwort. Der Server startet daraufhin das in der Anfrage angegebene CGI-Programm, welches die Anfrage verarbeitet, die gewünschte Antwort generiert und über den Server an den Browser leitet.
Im Hypertext-Transfer-Protokol (HTTP) gibt es für Browser-Anfragen zwei Kommandos: mit dem GET-Kommando können Daten nur direkt in der URL versendet werden. Da eine URL jedoch in ihrer Länge beschränkt ist, reicht dies nicht immer aus. Mit dem POST-Kommando lassen sich dagegen beliebig große Datenmengen übertragen.

JavaFSM nutzt zum Laden und Speichern getrennte CGI-Programme. Beim Laden sendet das Applet als Parameter den Automaten-Namen und ein Paßwort. Das CGI-Programm sucht den angegebenen Automaten und überprüft das zugehörige Paßwort. Tritt kein Fehler auf, sendet es die Daten des Automaten an das Applet. Zum Speichern von Automaten sendet das Applet ebenfalls Name und Paßwort, gefolgt von den Daten. Sofern der Name noch nicht vergeben ist, bzw das Paßwort stimmt, schreibt das CGI-Programm Paßwort und Daten in ein File. Die Antwort an das Applet besteht aus einer Bestätigung bzw. einer Fehlermeldung.
Das Paßwort dient hauptsächlich zum Schutz vor ungewolltem Überschreiben durch andere. So kann nur der Designer des Automaten auf das File zugreifen.

Über Applet-Parameter können bestehende Automaten als Beispiele in das Menü mit aufgenommen und von dort geladen werden.

## 5.6 Parser

Der Parser wird benötigt, um die logischen Ausdrücke der Übergangsbedingungen und Ausgabefunktionen (nur bei Mealy) auszuwerten.
Basis-Einheiten eines Ausdrucks sind die Operatoren UND (&), ODER (|) und NICHT (!), die Namen der Inputs als Eingangsvariablen und die Werte 0 und 1, sowie Klammern. Die Grammatik des Parsers lautet [St96]:

**programm:**
END
expression END
**expression:**
expression | term
term
**term:**
term **&** primary
primary
**primary:**
number 0,1
name Inputs
**(**expression**)**
**!** primary

Die Syntaxanalyse geschieht im Stil des rekursiven Abstiegs (top down). Jeder Produktionsregel der Grammatik entspricht eine Funktion, die andere Funktionen aufruft. Terminale Symbole (0, 1, Namen der Inputs) werden von den Syntaxanalyse-Funktionen expr(), term() und prim() erkannt. Sobald beide Operanden eines (Unter-) Ausdrucks bekannt sind, wird dieser ausgewertet.
Der Parser verwendet eine Funktion get_token() für das Einlesen der Eingabe. Diese liest jeweils die nächste Basis-Einheit des Ausdruck ein und speichert diese in einer globalen Variablen. Vor jedem Aufruf einer Parser-Funktion wird das nächste Token von get_token() ermittelt. Stößt get_token() auf einen Namen, wird dieser mit der Funktion look() in dem Vektor der Inputs gesucht und der aktuelle Wert ermittelt.
Bei Aufruf des Parsers wird der Ausdruck um ein Return erweitert, welches zur Erkennung des Endes dient. Findet get_token() dieses Zeichen, wird der Auswertungsvorgang beendet und der Wahrheitswert des Ausdrucks zurückgeliefert. Tritt während des Auswertungsvorgangs ein Fehler auf, wird eine BadExpressionException geworfen, welche den Parser abrupt beendet.

## 5.7 Export von Automaten nach VHDL / KISS

Um entworfene Automaten weiterverarbeiten zu können, bietet JavaFSM die Möglichkeit, diese in die gängigen Formate VHDL und KISS zu konvertieren. Die Ausgabe des VHDL- / KISS-Codes erfolgt in ein Textfeld, von wo aus er mittels "Copy&Paste" lokal weiterverarbeitet werden kann.

### 5.7.1 VHDL

VHDL ist eine inzwischen weltweit akzeptierte Hardware-Beschreibungssprache. Entwickelt wurde sie ursprünglich zur Dokumentation von Schaltkreisen: In Form von (lesbaren) Verhaltensbeschreibungen wird das nach außen sichtbare Verhalten definiert. Dabei kommen High-Level-Sprachkonstrukte wie IF-THEN-ELSE oder CASE zum Einsatz.

Beim Schaltungsentwurf generieren sog. Synthese-Tools daraus fertige Schaltnetze bzw. -werke in Form von Gatternetzlisten. Diese können auf verschiedene Ziele (Zeit, Platz, ...) optimiert sein.
Da VHDL von einer Vielzahl von Tools unterstützt wird, dient es auch dem Datenaustausch. Front-End Tools generieren dabei auf Wunsch Compiler gerechten VHDL-Code, der von anderen Tools weiterverarbeitet werden kann [LWS 94].
Um mit JavaFSM erstellte endliche Automaten praktisch nutzen zu können, wird auch hier eine Export-Möglichkeit nach VHDL angeboten.

### 5.7.2 KISS / PLA Design Format

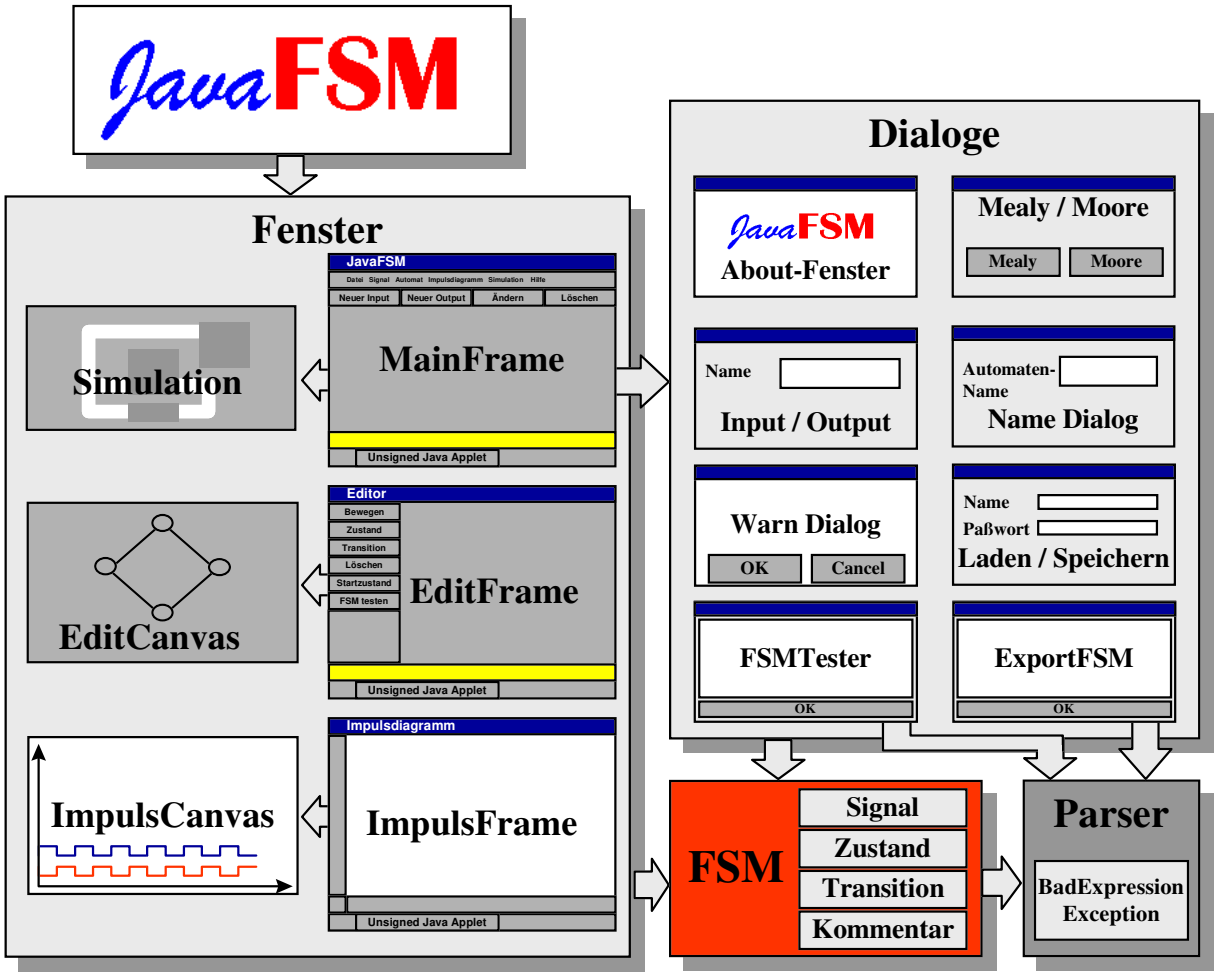
KISS kommt hauptsächlich in Verbindung mit programmierbarer Logik zum Einsatz. Es ist ein Technologie-unabhängiges Format, welches auf Wahrheitstabellen basiert. Neben diversen Features wie "don't cares", "subsets" etc. werden auch endliche Automaten unterstützt. Ein Design-File besteht aus einem Header mit Grundinformationen und einem Body mit den eigentlichen Logiktabellen. Die von JavaFSM verwendete Syntax ist in der folgenden Tabelle beschrieben:

| | |
|---|---|
| # | Kommentarzeile |
| .design name | Name des Designs |
| .inputnames name_1, [name_2 ...] | Hier werden alle Eingänge definiert (inkl. Clock- und Reset-Signal). In dieser Reihenfolge werden sie auch in der Wertetabelle aufgeführt. |
| .outputnames name_1, [name_2 ...] | Hier werden alle Ausgänge definiert. In dieser Reihenfolge werden sie auch in der Wertetabelle aufgeführt. |
| .clock signal sense | Einer der Eingänge wird hier als Clock-Signal definiert<br>signal: Name des Eingangs<br>sense kann die Werte "rising_edge" oder "falling_edge" annehmen (Flip-Flops werden bei steigender oder fallender Flanke des Clock-Signals aktiviert) |
| .async_reset signal sense state | Einer der Eingänge wird hier als Reset-Signal definiert<br>signal: Name des Eingangs<br>sense kann die Werte "rising_edge" oder "falling_edge" annehmen (s.o.)<br>state: Zustand, der nach einem Reset angenommen werden soll (Startzustand) |
| 010 z1 z2 01 | ein Zeile der Wertetabelle:<br>Eingangsbelegung<br>in dieser Zeile betrachteter Zustand<br>Zustand, in den der Automat bei dieser Konfiguration wechselt<br>Ausgangsbelegung in dieser Konfiguaration |
| 01- z1 z2 01 | Ein "-" in der Eingangsbelegung markiert ein "don't care". |

In JavaFSM wird von einer konkreten Zustandskodierung abstrahiert. Wird eine bestimmte Kodierung benötigt (z.B. als direkte Ausgabe), läßt sich diese in KISS wie folgt nachträglich definieren:

| .encoding | Zustandskodierung |
|---|---|
| state_name encoding | state_name: Zustand<br>encoding: numerischer Wert:<br>2#1010 binär<br>10#10 dezimal<br>16#A hexadezimal |

## 5.8 Beschreibung der Klassen




Basisklasse für das Programm JavaFSM ist sowohl für das Applet, wie auch für die Applikation die Klasse **JavaFSM**. Dazu erbt sie zum einen von java.applet.Applet, enthält aber zum anderen auch eine main()-Methode. Inhaltlich wird in dieser Klasse zuerst der **MMDialog** (Moore/Mealy Auswahl) und dann das Hauptfenster (MainFrame) geöffnet.

**MainFrame** ist das Hauptfenster, von dem aus das Programm gesteuert wird. Im Konstruktor dieser Klasse werden Instanzen fast aller anderer Klassen (Fenster, Dialoge, FSM, etc.) erzeugt. Das Fenster selbst enthält neben einem Menü und einiger Buttons vor allem die Simulations-Skizze, die von der Klasse **Simulation** (Panel) gezeichnet wird.

**FSM** ist ein Automaten-Objekt, das alle Subkomponenten des Automaten zusammenhält und eine grundlegende Simulations- und Editorfunktionalität bereitstellt. Komponenten eines Automates sind die **Signale**, **Zustände**, **Transitionen** und **Kommentare**, welche durch eigene Objekte (Klassen) repräsentiert werden. Zur Simulation benötigt FSM außerdem noch einen **Parser**, welcher ebenfalls als eigene Klasse realisiert wurde. Da die auszuwertenden logischen Ausdrücke von den Eingangssignalen abhängen dürfen, benötigt der Parser neben dem eigentlichen Funktionsstring auch den Vector mit den Inputs, um deren aktuellen Werte ermitteln zu können.

**EditFrame** ist das Editor-Fenster, das neben einiger Buttons, etc. ebenfalls eine Zeichenfläche (**EditCanvas)** enthält. Da die Klasse FSM die wichtigsten Editorfunktionen zur Verfügung stellt, beschränkt sich die hier implementierte Funktionalität auf GUI-Komponenten. EditCanvas stellt eine Zeichenroutine zur Verfügung, mit der ein Automat skaliert gezeichnet werden kann. Diese dient im Simulations-Objekt zur Darstellung des Automaten im Delta-Schaltnetz.

Das dritte Fenster ist das **ImpulsFrame**, in dem das Impulsdiagramm gezeichnet wird. Dazu besitzt es neben zwei Scrollbalken ebenfalls eine Zeichenfläche **ImpulsCanvas**. Generiert wird das Impulsdiagramm aus einer in FSM gespeicherten Zustands- und in den Signalen gespeicherten Wertefolge.

Schließlich besitzt JavaFSM noch eine Reihe weiterer Dialog-Fenster, die zum Teil wichtige Funktionen erfüllen. Der **MMDialog** fragt zu Beginn des Programms ab, ob es sich um einen Mealy- oder Moore-Automaten handeln soll. Der **About**-Dialog zeigt Informationen über das Programm an. Der **SignalDialog** wird zum Generieren und Ändern von Signalen verwendet. Im **NameDialog** kann der Automaten-Name geändert werden. Der **WarnDialog** erscheint beim Verlassen des Simulationsmodus, bevor die Simulationsdaten gelöscht werden. Der **UserDialog** wird beim Laden bzw. Speichern von Automaten im Applet-Modus benutzt und fragt Filenamen und Paßwort ab. **FSMTester** wird vom Editor benutzt und überprüft den Automaten auf Fehler. Dabei werden rekursiv alle möglichen Belegungen getestet, um sicherzustellen, daß kein Nicht-Determinismus auftritt. Hierzu bekommt FSMTester eine Referenz auf den Parser übergeben. Die Ergebnisse werden in einer Text Area ausgegeben. **ExportFSM** schließlich konvertiert einen Automaten nach VHDL oder KISS und schreibt das Egebnis ebenfalls in eine TextArea. Auch hier wird eine Referenz auf den Parser übergeben. Im Gegensatz zu VHDL müssen beim KISS-Export, ähnlich wie beim FSMTester, rekursiv alle möglichen Belegungen geprüft werden, wodurch diese Konversion länger dauern kann.

# 6 Bedienungs-Anleitung

## 6.1 JavaFSM als Applikation

Zum Ausführen von JavaFSM benötigen Sie das JDK von Sun. Laden Sie sich die Datei JavaFSM.zip herunter und entpacken Sie diese. Sie enthält neben dem Programm JavaFSM auch einige Beispielautomaten, sowie Dokumentation. Wechseln Sie in das Verzeichnis JavaFSM und starten Sie den Interpreter wie folgt:

**"java -classpath PFAD DE.uni_hamburg.informatik.tech.JavaFSM.JavaFSM"**, wobei der Pfad auf das Verzeichnis JavaFSM zeigen muß.

Sie können unter Windows 95 / NT auch die mitgelieferte Batch-Datei "JavaFSM.bat" anpassen. Sofern sich das Verzeichnis JavaFSM nicht direkt auf dem Laufwerk C: befindet,mssen Sie hier den Pfad angeben. Außerdem muß der Pfad auf das JDK gesetzt sein.

## 6.2 Auswahl zwischen Mealy- und Moore-Schaltwerken

Beim Starten des JavaFSM erscheint ein Dialogfenster zur Auswahl des Schaltwerk-Typs. Bei Moore-Schaltwerken hängen die Ausgabewerte nur von der Berechnung des Automaten ab. Bei Mealy hingegen werden die Eingabewerte mit in das Lambda-Schaltnetz geführt, sodaß die Ausgabewerte auch direkt von den Eingabewerten abhängen können und sich somit auch zwischen den Takten ändern können.
Über den Menüpunkt "Automat-Moore" bzw "Automat-Mealy" kann der Schaltwerktyp geändert werden. Dies ist jedoch nur im Editiermodus möglich, da der Automat während der Simulation nicht verändert werden darf. Beim Wechsel vom Mealy-Schaltwerk zum Moore-Schaltwerk gehen dabei komplexe Ausgabefunktionen verloren und werden durch "0" ersetzt.

## 6.3 Speichern und Laden von entworfenen Automaten

JavaFSM bietet die Möglichkeit, selbst entworfene Automaten zu speichern und wieder zu laden. In der Applikation läßt sich dabei über die Menüpunkte Laden und Speichern ein Dateiname auswählen. Zur einheitlichen Kennzeichnung sollten diese die Endung ".fsm" tragen. Hierbei ist zu beachten, daß der CLASSPATH gesetzt sein muß!

Applets sind aus Sicherheitsgründen jegliche Zugriffe auf den Client-Rechner verboten und es ist daher nicht möglich, Daten direkt beim Anwender zu speichern.
Bei Auswahl des Menüs "Datei-Laden" / "Datei-Speichern" im Applet öffnet sich ein Dialogfenster. Dateien erhalten einen Namen unter dem sie gespeichert werden und ein Paßwort, das ein versehentliches Überspeichern verhindern soll und Daten vor anderen Anwendern schützt. Dabei ist zu beachten, daß auf dem Server alle Automaten im gleichen Verzeichnis abgelegt werden. Daher sollte ein Automat einen eindeutigen Namen erhalten (username_automatenname). Nach Eingabe des Filenamens und des Paßwortes und Bestätigung werden die Daten zum Server übertragen. Eventuell auftretende Fehler werden in der Statuszeile ausgegeben. Wurde der Automat ordnungsgemäß geladen / gespeichert, schließt sich das Dialogfenster. Wird beim Speichern ein bereits vorhandener Filename gewählt, muß das zugehörige Paßwort angegeben werden. Sonst erscheint die Fehlermeldung „Dateiname vergeben / Paßwortfehler“. Wählen Sie dann bitte einen anderen Filenamen.

### 6.3.1 Firewall / Proxy

Ein Java Applet darf aus Sicherheitsgründen eine Netzwerkverbindung nur zu dem Server herstellen, von dem es geladen wurde. Liegt zwischen dieser Verbindung ein Proxy-Server (Firewall), können leider keine Automaten geladen bzw. gespeichert werden, da der Proxy selbst nicht kontaktiert werden kann! Wer dennoch mit JavaFSM arbeiten möchte, kann es sich downloaden und als Applikation starten.

## 6.4 Laden von Beispielautomaten

Unter dem Menüpunkt "Datei-Beispiele" befinden sich einige Beispielautomaten, die bei Auswahl vom Webserver geladen werden. Sollten beim Laden irgendwelche Fehler auftreten, werden diese in der Statuszeile ausgegeben. Diese Auswahl ist in der Applikation nicht möglich. Hier können die Beispiele direkt aus dem Unterverzeichnis Beipiele geladen werden.

## 6.5 Festlegen des Automaten-Namen

Über das Menü Automat - Automaten-Name läßt sich ein Name für den Automaten festlegen. Dieser erscheint dann in den Titelleisten der Fenster.

## 6.6 Festlegen der Ein-/ und Ausgänge

Über die Buttons bzw. das Menü Signal können die Eingänge und Ausgänge eingegeben, geändert bzw. gelöscht werden.
**Neu:** Es öffnet sich ein Dialogfenster, indem man den Namen des Inputs / Outputs und bei Inputs den Initialwert festlegen können. Ein Name darf Buchstaben, Ziffern und die Zeichen "/" und "_" enthalten, wobei das erste Zeichen keine Ziffer sein darf.
**Ändern:** Nach der Auswahl des Ändern-Buttons oder der Menüs Ändern klickt man den zu ändernden Input / Output an. Es erscheint das Dialogfenster, indem man die Änderungen vornehmen können.
**Löschen:** Nach der Auswahl des Löschen-Buttons oder der Menüs Löschen klickt man den zu löschenden Input / Output an. Dieser wird dann gelöscht.

## 6.7 Entwurf des zugrundeliegenden Automaten mit Hilfe des Editors

Über das Menu Automat-Editor gelangt man in den Automaten-Editor.
Der linken Bereich enthält die Buttons und darunter das Eingabefeld. Rechts befindet sich die Zeichenfläche.
Der Editor arbeitet Modus-orientiert. Die Buttons haben dabei folgende Bedeutung:
**Zustände:** Durch Anklicken des Buttons "Zustand" gelangt man in den Modus zum Einfügen der Zustände. Es können nun beliebig viele Zustände durch Klicken in die rechte Zeichenfläche eingegeben werden.
**Transitionen:** Durch Anklicken des Buttons "Transition" gelangt man in den Modus zum Einfügen der Transitionen. Zum Eingeben einer Transition werden die beiden zu verbindenden Zustände nacheinander ausgewählt. Beim Anklicken des ersten Zustandes färbt sich dieser rot. Danach wird der Folgezustand angeklickt. Die Transition wird eingefügt. Der Folgezustand bleibt zunächst ausgewählt (rot), so daß er schon der Ausgangszustand für die nächste Transition ist. Durch Klicken ins "Leere" wird er wieder deselektiert und ein anderer Ausgangszustand kann angeklickt werden.
**Löschen:** Durch Anklicken des Buttons "Löschen" gelangt man in den Löschmodus. Alles, was im Löschmodus angeklickt wird, wird gelöscht. Bei Zuständen werden auch die zugehörigen Transitionen gelöscht.
**Startzustand:** In diesem Modus kann man durch Anklicken den Startzustand festlegen. Es muß in jedem Automaten ein Startzustand festgelegt werden.
**Bewegen:** Im Bewegen-Modus können Zustände und Kommentare verschoben werden. Bei Zuständen verschieben sich die Transitionen dabei mit. Außerdem können Transitionen und Zustände ausgewählt werden, um im Eigenschaftsfeld die Übergangsbedingungen, die Ausgabefunktionen, sowie die Namen der Zustände festzulegen. Für einen Kommentar kann im Eigenschaftsfeld der Text festgelegt werden.

**Kommentar:** Hier lassen sich auf der Zeichenfläche Kommentare einfügen. Diese sind umrandet und dürfen mehrere Zeilen umfassen. Durch Anklicken des „Übernehmen“-Buttons wird der Text in den Kommentar übernommen.

## 6.8 Definition der logischen Übergangsbedingungen und Ausgabefunktionen und Ändern der Zustandsnamen

Beim Anklicken einer Transition (im Bewegen-Modus) erscheint unter den Buttons ein Textfeld, das die logische Übergangsfunktion enthält. Dabei gilt folgende Syntax:

- & : logisches AND
- | : logisches OR
- ! : logisches NOT

Bei von einem Zustand ausgehenden Transitionen kann eine der Übergangsfunktion auch nur den "*" enthalten. Der "*" steht für alle anderen Belegungen, d.h. diese Transition ist aktiviert, wenn keine andere von diesem Zustand ausgehende Transition aktiviert ist.

## 6.9 Überprüfung des Automaten auf Korrektheit

Durch Anklicken von "FSM testen" (im Editor) wird der aktuelle Automat auf Korrektheit überprüft. Eventuelle Fehler werden in einem Fenster ausgegeben. Bei komplexeren Automaten kann dies einige Minuten dauern (neben der größe des Automaten ist die Anzahl der Inputs entscheident für den Zeitbedarf). Dabei wird zunächst überprüft, ob überhaupt Zustände und Transitionen vorhanden sind, und ob einer der Zustände als Startzustand definiert wurde. Danach werden alle möglichen Belegungen durchgegangen und festgestellt, ob jeweils genau eine Transition aktiviert ist. In einem Dialogfenster werden dabei Statusmeldungen und gegebenenfalls Fehler ausgegeben

## 6.10 Zwischen Editier- und Simulationsmodus hin- und herschalten

Das Programm unterscheidet zwischen Editier- und Simulationsmodus. Im Editiermodus können Inputs / Outputs festgelegt und der Automat eingegeben werden. Im Simulationsmodus können Inputwerte verändert, Takte generiert und das Impulsdiagramm angezeigt werden. Den Modus wechselt man über das Menü Simulation-Simulieren bzw. Simulation-Editieren. Beim Wechsel vom Simulations- in den Editiermodus gehen alle bisherigen Simulationsdaten verloren.

## 6.11 Simulation durch Verändern der Eingangswerte und Taktgebung

Nachdem in den Simulationsmodus gewechselt wurde, können die Werte der Inputs durch Anklicken verändert werden. Durch Anklicken des Takt-Buttons oder über das Menü Simulation-Takt kann ein Takt generiert werden. Der Automat berechnet dabei den Folgezustand und wechselt in diesen. An den Ausgängen (Outputs) liegen nun die Werte der Ausgabefunktionen dieses Zustandes an. Durch Anklicken von Reset (oder Menü Simulation-Reset) geht der Automat in den Ursprungszustand zurück (Startzustand, Initialwerte der Inputs, Outputwerte der Ausgabefunktion des Startzustandes). Das Impulsdiagramm wird ebenfalls zurückgesetzt.

## 6.12 Ausgabe des Simulationsergebnisses im Impulsdiagramm

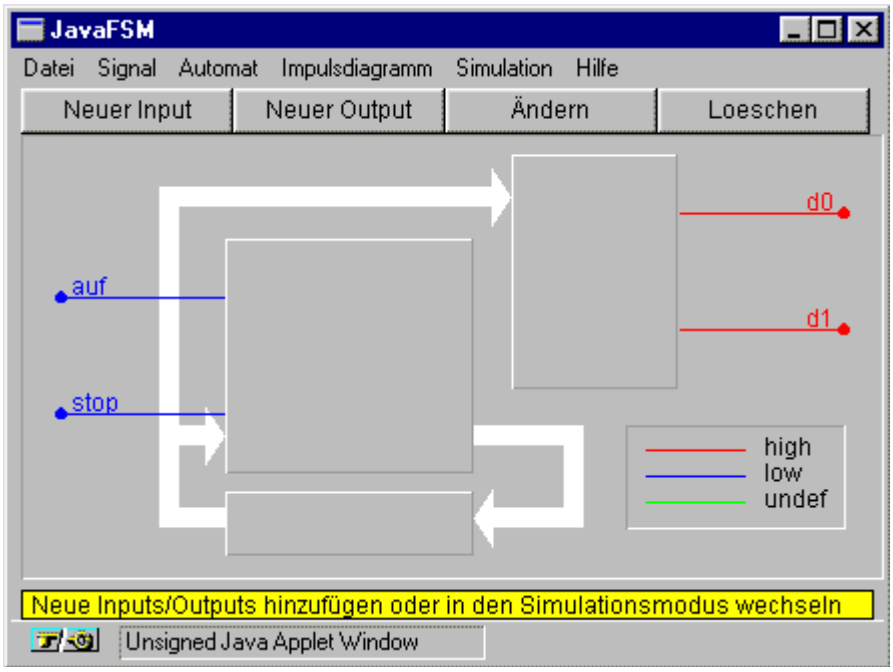
Über das Menü Impulsdiagramm-Anzeigen (nur im Simulationsmodus aktiv) öffnet sich das Impulsdiagramm.

### 6.13 Export des Automaten im VHDL/ KISS-Format

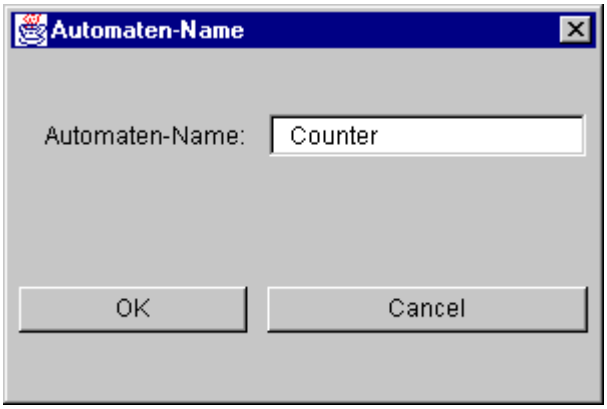
Über das Menü Datei-Konvertieren-VHDL bzw. Datei-Konvertieren-KISS läßt sich der Automat im VHDL bzw. KISS-Format ausgeben. Da ein Applet aus Sicherheitsgründen weder Drucken noch speichern kann, wird der Text in einem Textfeld ausgegeben, von wo er mit Copy und Paste in einem normalen Editor gespeichert werden kann.

# 7 Beispiel: Counter

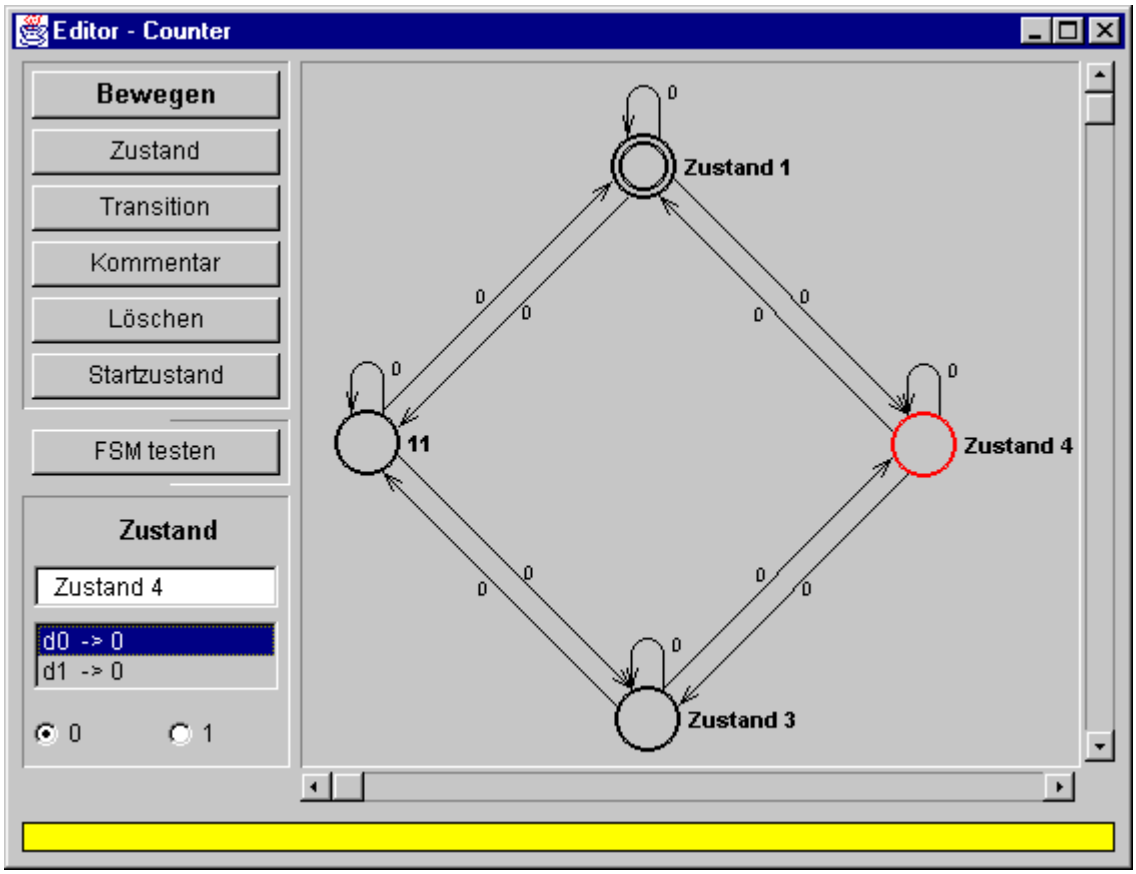
Um die Funktionsweise von JavaFSM zu erläutern, soll nun ein 2-bit-Auf- / Abwärtszähler konstruiert werden. Dieser ist unter dem Namen "Counter" in den Beispielen enthalten.
Da wir keine asynchronen Ausgaben benötigen, reicht ein Moore-Automat. Entweder wählen wir diesen schon beim Start von JavaFSM aus, oder wir stellen dies danach im Menü "Automat" ein.
Nun benötigen wir zwei Eingänge (Inputs) mit den Namen "auf" und "stop". Der erste bestimmt die Zählrichtung und der zweite hält den Zähler an. Außerdem benötigen wir die zwei Ausgänge (Outputs) mit Namen "d0" und "d1".



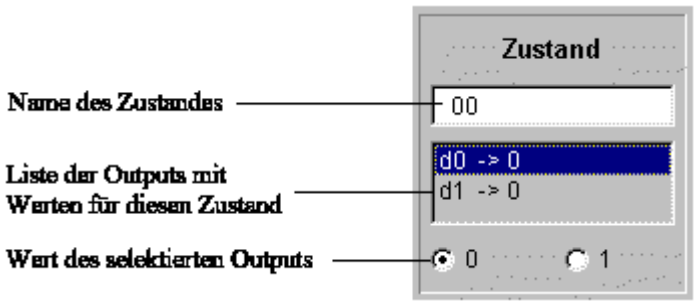

Über das Menü Automat - Automaten-Name können wir dem Automaten einen Namen geben. Dieser erscheint dann auch in der Titelleiste der Fenster.



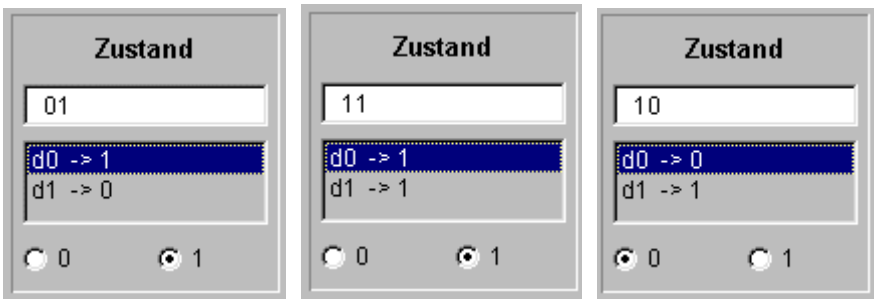
Über den Punkt "Editor" im Menü "Automat" gelangen wir in den Editor und geben dort den Automaten ein.
Zuerst wechseln wir mit dem "Zustand"-Button in den Einfügemodus für Zustände. Mit jedem Mausklick in die Zeichenfläche wird dort ein neuer Zustand eingefügt. Im "Bewegen"-Modus können die Zustände anschließend beliebig angeordnet werden. Generieren Sie so 4 Zustände.
Nun bilden wir zwei gegenläufige Kreise von Transitionen, indem wir zunächst mit dem "Transitionen"-Button in den entsprechenden Modus wechseln und dann von Zustand zu Zustand klicken. Dabei wird von einem vorher selektierten Zustand (rot markiert) eine Transition zu dem neu angeklickten Zustand eingefügt. War noch kein Zustand rot markiert, dann wird der Zustand lediglich selektiert.
Um stoppen zu können, braucht jeder Zustand noch eine Transition auf sich selbst (Schleife). Dazu klicken wir jeden Zustand zweimal an.



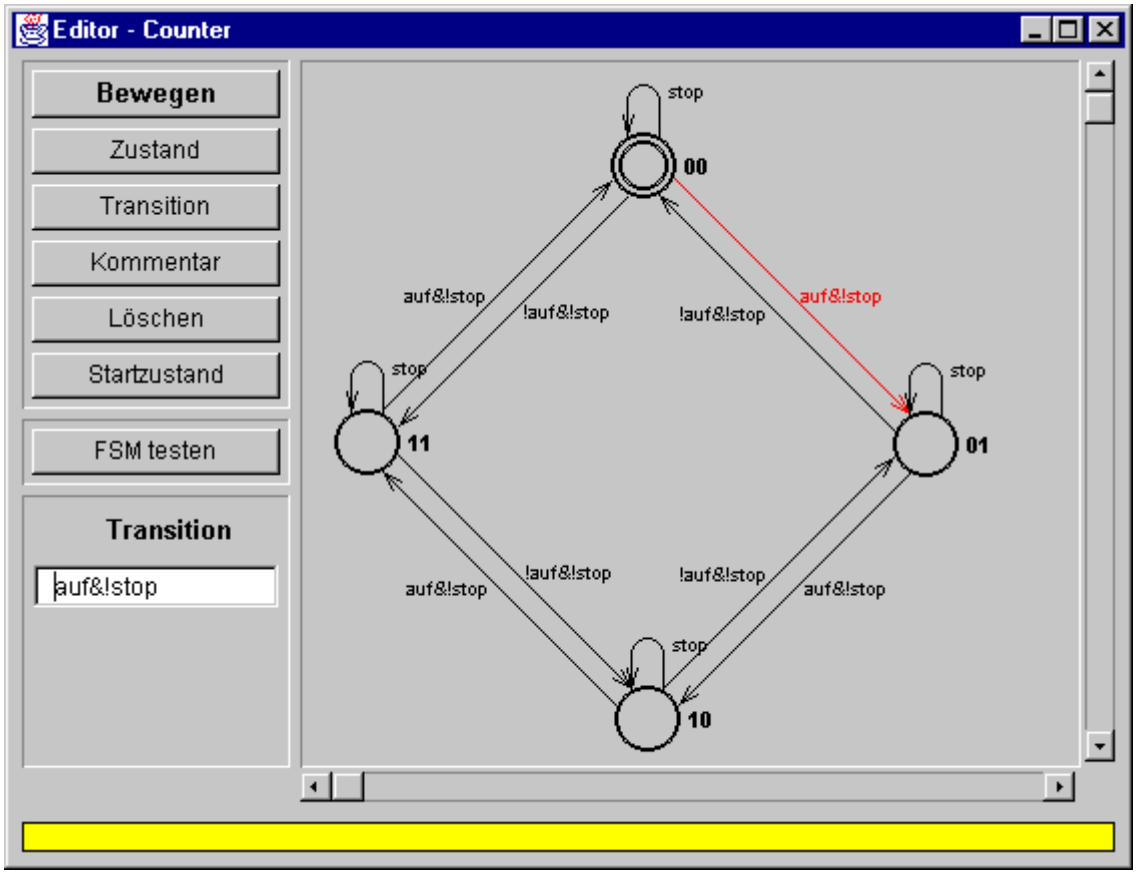

Um den Zuständen sinnvolle Namen zu geben, klicken wir diese im "Bewegen"-Modus an. Daraufhin werden unten links im Fenster die Eigenschaften des Zustandes eingeblendet. Im Eingabefeld kann der Name geändert werden. Mit einem "**Return**" wird diese Änderung übernommen. In unserem Beispiel nennen wir die Zustände "00", "01", "10" und "11".



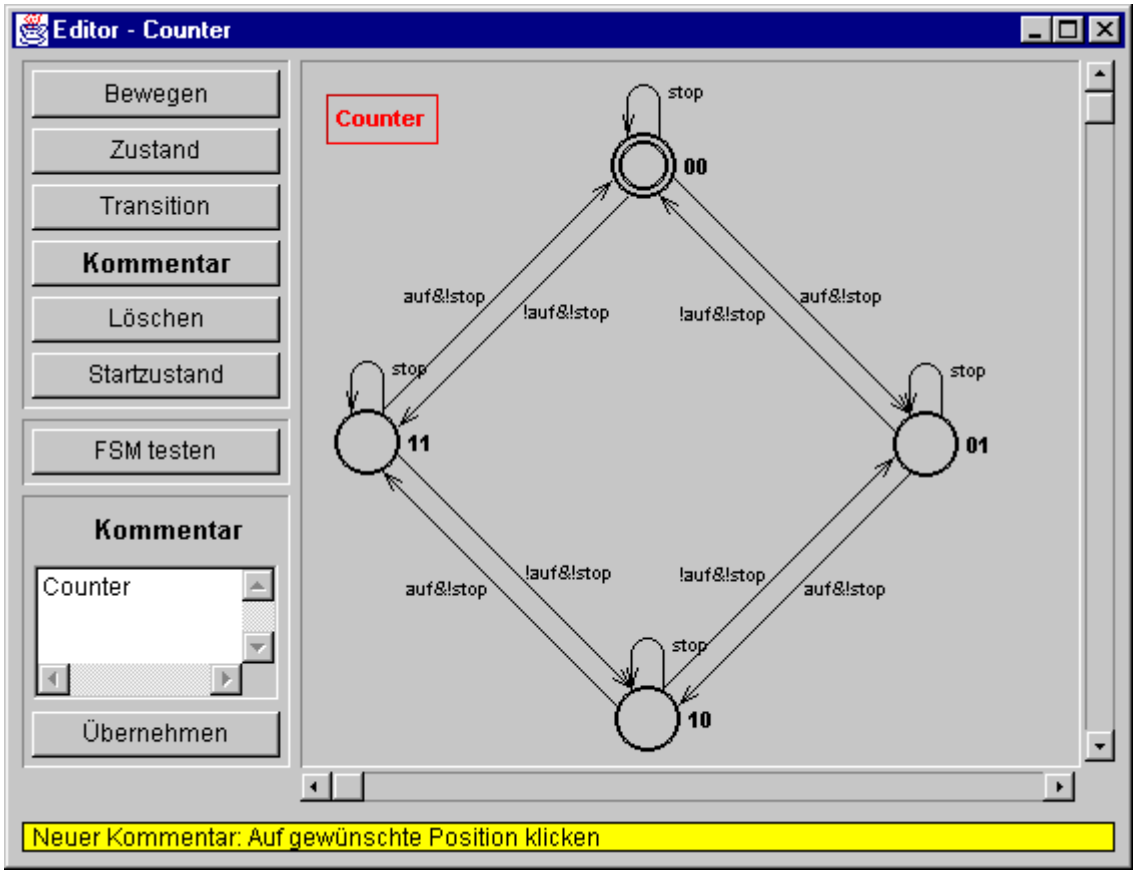

Außerdem können in dem Eigenschaftsfeld die Ausgabewerte des Zustandes (für d0 und d1) festgelegt werden. Dazu werden die entsprechenden Outputs in der Liste ausgewählt und der Wert eingestellt:



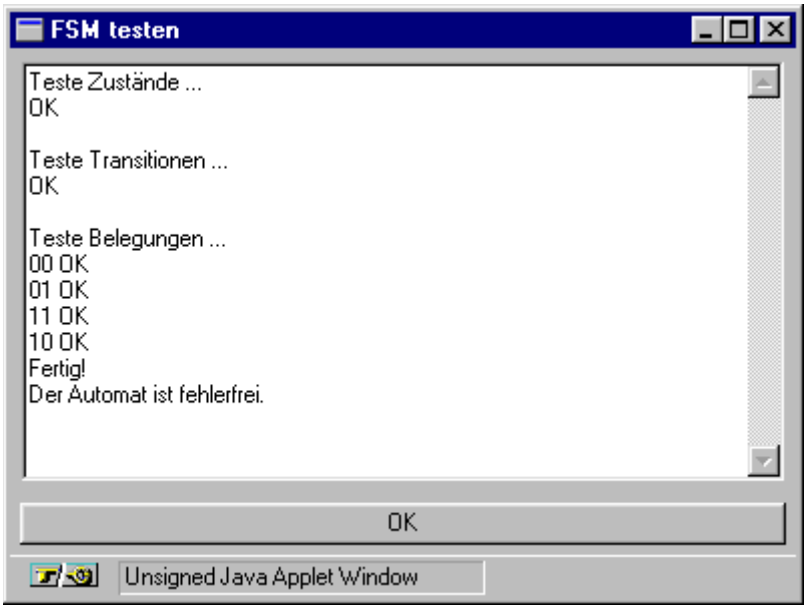

Da es sich in diesem Beispiel um einen Moore-Automaten handelt, kann hier nur "0" oder "1" angegeben werden. Bei Mealy-Automaten ist es an dieser Stelle auch möglich, komplexere Funktionen anzugeben.
Anschließend werden die Übergangsbedingungen der Transitionen festgelegt. Durch Anklicken der Transitionen (im "Bewegen"-Modus) erscheint im Eigenschaftsfeld ein entsprechendes Eingabefeld.
**Auch hier werden Änderungen erst nach einem "Return" übernommen.**
Für Transitionen, die im Uhrzeigersinn verlaufen, geben wir "auf&!stop" ein, für Transitionen, die entgegen des Uhrzeigersinns verlaufen, geben wir "!auf&!stop" ein. Für Schleifen lautet die Übergangsbedingung "stop".
Jetzt wählen wir noch einen Zustand als Startzustand aus, indem wir zuerst den "Startzustand"-Button und danach den Zustand "00" anklicken.



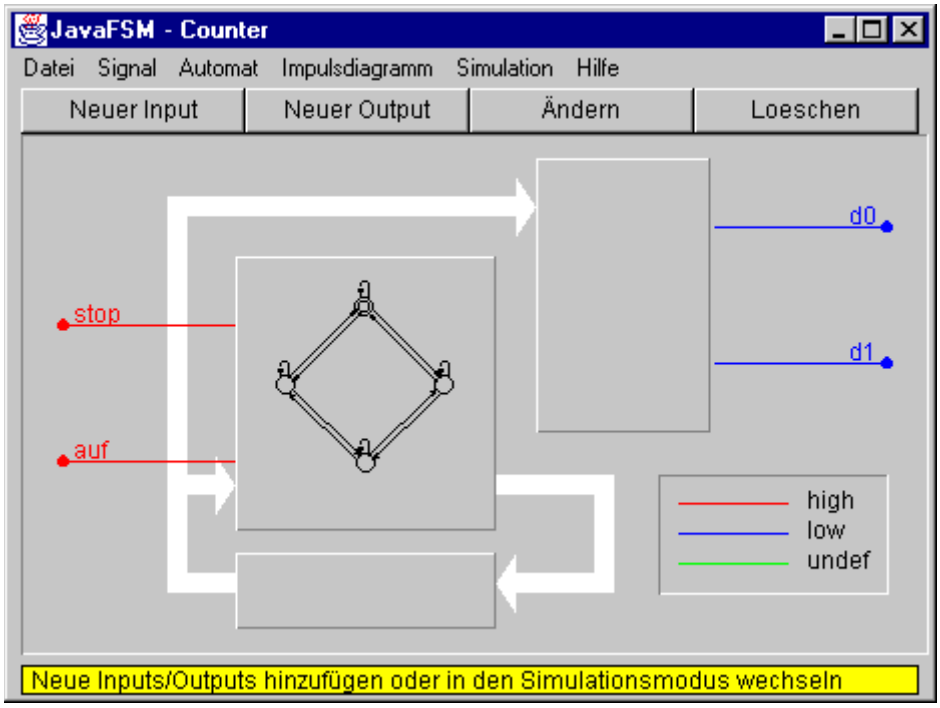

Zusätzlich besteht die Möglichkeit, den Automaten mit Kommentaren zu versehen. Dabei wechselt man in den Kommentar-Modus und fügt den Kommentar ein. Im Eigenschaftsfeld kann dann der Text für den Kommentar eingegeben und durch Anklicken des Buttons übernommen werden.



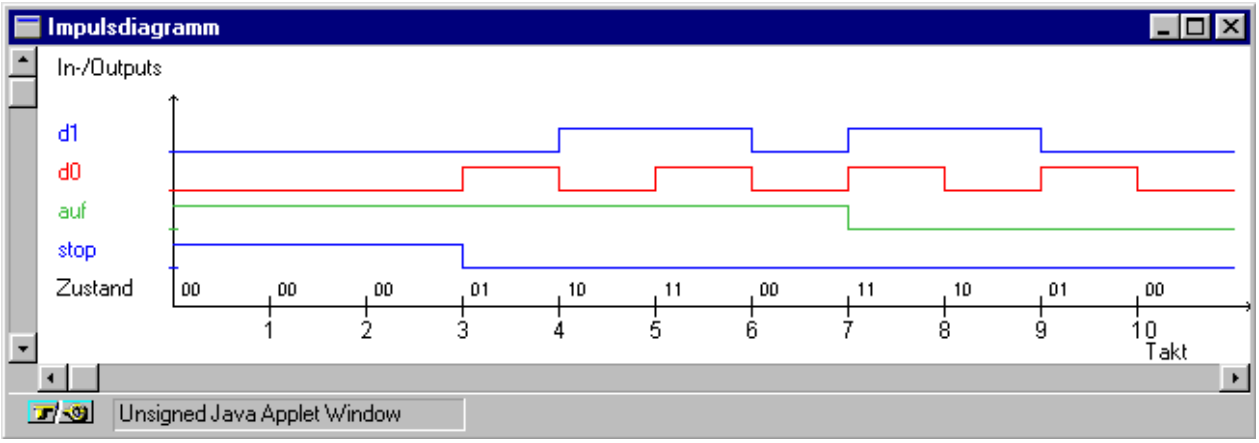

Abschließend können wir den Automaten noch mit "FSM testen" auf Korrektheit überprüfen. Dabei sollte es zu folgender Ausgabe kommen:




Wir wechseln jetzt in den Simulationsmodus, indem wir im Hauptfenster den Menüpunkt "Simulieren" im Menü "Simulation" anwählen. Im Delta-Schaltnetz ist nun der Automat zu sehen. Der aktuelle Zustand und die momentan aktive Transition sind rot markiert. Durch Anklicken können wir die Werte der Inputs ändern. Mit einem Klick auf den "Takt"-Button wird der Folgezustand berechnet und der Automat wechselt in diesen über. Die Outputs nehmen die in diesem Zustand festgelegten Werte an.




Über "Anzeigen" im Menü "Impulsdiagramm" kann man sich das Impulsdiagramm anzeigen lassen.

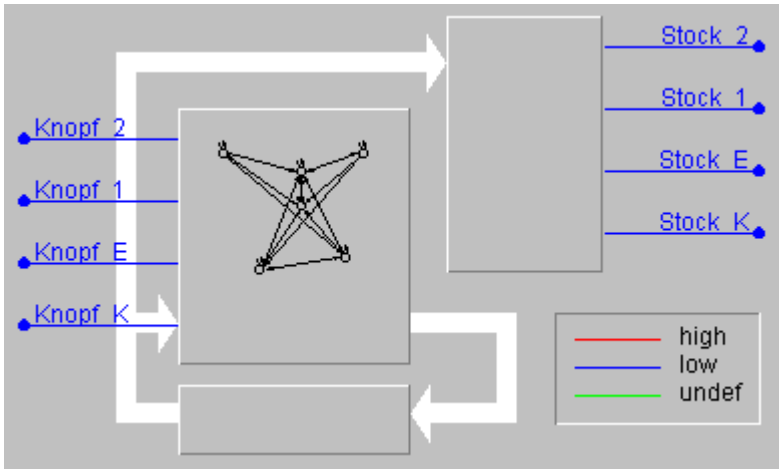



Zum Speichern des Automaten im ruft man im Menü Datei den Punkt Speichern auf.
In der Applikation öffnet sich ein Datei-Fenster, in dem man die zu speichernde Datei angeben kann.
Im Applet gibt man in einem Dialogfenster für den Automaten einen Filenamen an. Dabei ist zu beachten, daß alle Automaten im gleichen Verzeichnis auf dem Server gespeichert werden. Daher sollte bei der Namensgebung eine eindeutige Bezeichnung gewählt werden (username_automatenname). Zusätzlich wird ein Paßwort vergeben. Dies schützt vor Überschreiben durch andere Anwender. Nur mit dem richtigen Paßwort kann eine bestehende Datei überschrieben werden. Wird ein bereits verwendeter Dateiname mit einem falschen Paßwort gewählt, so kommt es zu der Fehlermeldung "Datei vorhanden / falsches Paßwort".
Eine gespeicherte Datei kann später über das Menü Datei - Laden und Angabe von Dateiname und Paßwort wieder geladen werden.

In JavaFSM besteht die Möglichkeit, entworfene Automaten nach VHDL und KISS zu exportieren. Dazu wählt man im Menü Datei / Export das gewünschte Format. Leider haben Applets keine Möglichkeit, auf lokale Dateien zuzugreifen. Die Ausgabe erfolgt daher in einem seperaten Fenster, von wo sie mittels "Copy und Paste" weiterverarbeitet werden kann.

# 8 Ein komplexeres Beispiel: Fahrstuhl

Der Fahrstuhl, welcher sich unter den Beispielen befindet, modelliert einen Fahrstuhl, der 4 Stockwerke bedient.
Als Eingabewerte dienen hier vier Knöpfe für die jeweiligen Stockwerke, wobei es in diesem Fall egal ist, ob jemand im Fahrstuhl den Knopf 2. Stock drückt, oder im 2. Stock steht und den Fahrstuhl ruft.
Als Ausgabewerte haben wir vier Signale, die angeben, in welchem Stockwerk sich der Fahrstuhl gerade befindet.




**Der Automat enthält 6 Zustände:**

| Zustand | Beschreibung | K | EG | 1 | 2 |
|---|---|---|---|---|---|
| Keller | gibt an, daß sich der Automat im Keller befindet | 1 | 0 | 0 | 0 |
| Erdgeschoß vu | gibt an, daß sich der Automat von unten kommend im Erdgeschoß befindet | 0 | 1 | 0 | 0 |
| Erdgeschoß vo | gibt an, daß sich der Automat von oben kommend im Erdgeschoß befindet | 0 | 1 | 0 | 0 |
| 1. Stock vu | gibt an, daß sich der Automat von unten kommend im 1. Stock befindet | 0 | 0 | 1 | 0 |
| 1. Stock vo | gibt an, daß sich der Automat von oben kommend im 1. Stock befindet | 0 | 0 | 1 | 0 |
| 2. Stock | gibt an, daß sich der Automat im 2. Stock befindet | 0 | 0 | 0 | 1 |

Befindet sich der Fahrstuhl im Erdgeschoß oder 1. Stock, so soll er, sofern mehrere Knöpfe gedrückt sind, zunächst in die gleiche Richtung weiterfahren. Daher wurden für diese Stockwerke jeweils zwei Zustände eingefügt, die angeben aus welcher Richtung der Fahrstuhl kommt.
**Übergangsbedingungen:**

Wenn nur ein Knopf gedrückt ist, soll der Fahrstuhl natürlich in dieses Stockwerk fahren. Sind mehrere Knöpfe gedrückt, so soll er seine Fahrtrichtung beibehalten und das nächstliegende Stockwerk anfahren.

| **von** | **nach** | **wenn** |
|---|---|---|
| Keller | Erdgeschoß vu | Knopf_EG |
| Keller | 1. Stock | !Knopf_EG&Knopf_1 |
| Keller | 2. Stock | !Knopf_EG&!Knopf_1&Knopf_2 |
| Erdgeschoß vu | Keller | !Knopf_2&!Knopf_2&Knopf_K |
| Erdgeschoß vu | 1. Stock | Knopf_1 |
| Erdgeschoß vu | 2. Stock | !Knopf_1&Knopf_2 |
| u.s.w. | | |

# 9 Kritische Betrachtung

In diesem Kapitel sollen einige Schwächen von JavaFSM beleuchtet werden.

JavaFSM stellt eine Transition als geraden Pfeil zwischen den Zuständen dar. Bei vielen Transitionen kann dies unübersichtlich werden. Hier könnte die Einführung von Bezier-Kurven abhilfe leisten. Da dies jedoch einen erheblichen Mehraufwand bedeutet, der auf die Funktionalität keinen Einfluß hat, haben wir im Rahmen dieser Arbeit darauf verzichtet.

Da der Editor bei jeder Veränderung den gesamten Automaten neu zeichnet, tritt beim Verschieben von Elementen ein zum Teil erhebliches Flackern auf. Verstärkt wird dies durch die langsame Textausgabe von Java, die bei der Beschriftung von Zuständen, Transitionen und Kommentaren benutzt wird. Die Benutzung des XOR-Zeichenmodus führt zu keinem zufriedenstellenden Ergebniss, da es bei den mehrfarbigen Zeichnungen in JavaFSM zu Falschfarben kommt.

Beim KISS-Export und beim FSM-Testen werden rekursiv alle möglichen Belegungen durchgetestet, wodurch erheblicher Rechenaufwand entsteht. Eine besondere Rolle spielen hierbei die Inputs, da mit ihrer Anzahl der Rechenaufwand exponentiell ansteigt. Bei kleineren Automaten (Counter, Fahrstuhl, etc.) geschieht dies noch in akzeptabler Zeit, wogegen es bei komplexeren Automaten schnell mehrere Minuten dauern kann. Abhilfe könnten eventuell Binary Decision Diagrams schaffen (siehe Ausblick), von deren Einsatz im Rahmen dieser Arbeit jedoch abgesehen wurde.

Da Java eine junge Programmiersprache ist, die sich noch in der Weiterentwicklung befindet, treten zum einen Schwächen in der Sprache selbst auf. Zum Anderen muß die Virtual Machine für jede Plattform und jeden Browser getrennt implementiert werden. Obwohl SUN diese genau spezifiziert hat, treten implementations-abhängige Unterschiede und Fehler auf.

Java ist eine junge und noch nicht voll ausgereifte Programmiersprache. Außerdem wird die Java Virtual Machine für jede Plattform (und jeden Browser) getrennt implementiert. Daher treten auf den verschiedenen Plattformen hin und wieder unterschiedliche Fehler auf.

So machen die Scrollbalken auf einigen Plattformen Schwierigkeiten. Zum Beispiel sendet Netscape beim Verschieben der Scrollbalken ständig Events, so daß das Panel / Canvas ständig neu gezeichnet wird und es zu einem starken Flackern kommt. In Version 1.1 des JDK wird ein Scrollpanel in der Klassenbibliothek enthalten sein.
Es traten weiterhin Probleme beim Repaint der Fenster auf. Dies scheint bei Windows95 mit installiertem Plus-Paket besonders der Fall zu sein.

# 10 Zusammenfassung und Ausblick

Mit JavaFSM können endliche Automaten zum Entwurf von Schaltwerken eingegeben, getesten und simuliert werden. Durch den Export der Automaten nach VHDL und KISS ist eine Weiterverwendung möglich. Die Implementation als Applet und Applikation und die Plattformunabhängigkeit von Java ermöglichen eine weite Verbreitung von JavaFSM, die insbesondere beim Einsatz in der Lehre wichtig ist.

Da JavaFSM auf dem JDK 1.0.2 basiert, nutzt es spezielle Fähigkeiten von Java ab Version 1.1 nicht aus. So ist zum Beispiel zu prüfen, ob und in welcher Form es möglich ist, Automaten auch auszudrucken. Ferner können signierten Applets mehr Zugriffsrechte eingeräumt werden, so daß es möglich wird, Automaten auch lokal zu speichern.

Es kann eventuell gewünscht sein, neben dem Automaten auch die Simulationsdaten zu speichern. Hierzu muß das Automatenformat zur Speicherung entsprechend ergänzt werden.

Es erscheint wünschenswert, die Anordnung der In- / Outputs im Impulsdiagramm verändern zu können, um zusammengehörige Werte zu gruppieren. Hierbei ist zu prüfen, wie so eine Anordnung durchgeführt werden kann.

Außerdem kann es sinnvoll sein, Signale in Vektoren zusammenzufassen, um zum Beispiel Datenbusse zu gruppieren und bei der Simulation statt der einzelnen Pegel den Wert des Busses zeigen bzw. eingeben zu können.

Die Objektorientierung legt die Verwendung von hierarchischen Automaten nahe. Hier verbirgt sich hinter einem Zustand wieder ein eigenständiger Automat. Dadurch wird eine Kapselung logischer Einheiten möglich, die besonders bei komplexeren Automaten wünschenswert erscheint.

Schließlich bleibt noch zu prüfen, welche Vor- und Nachteile Binary-Decision-Diagramms (BDD) gegenüber des in JavaFSM verwendeten "just-in-time"-Parsing bieten würden. Im Rahmen dieser Arbeit fiel die Entscheidung gegen BDD, da das erreichte Ergebnis zufriedenstellend war und wir Probleme beim Handling großer Datenmengen innerhalb der Virtual-Machine sahen. Hinzu kommt der (wahrscheinlich) vergrößerte Rechenaufwand, der in JavaFSM bewußt vermieden wurde.

# 11 Literaturverzeichnis


[La87] Klaus Lagemann, Rechnerstrukturen, Springer Verlag, Heidelberg, 1987

[LWS94] Gunther Lehmann, Bernhard Wunder, Manfred Selz, S*chaltungsdesign mit VHDL,* Franzis´ Verlag GmbH, Poing, 1994

[HU93] John E. Hoppcroft, Jeffrey D. Ullman, *Einführung in die Automatentheorie, formale Sprachen und Komplexitätstheorie,* Addison Wesley Publishing Company, Inc., 1993

[LP96] Laura Lemay, Charles L. Perkins, *Teach Yourself Java in 21 Days,* Sams.net Publishing, 1996

[CH96] Garry Cornell, Cay S Horstmann, *Core Java,* Prentice Hall PTR, 1996

[Fla96] David Flanagan, *Java in a Nutshell,* O´Reilly & Associates, Inc, 1996

[AG96] Ken Arnold und James Gosling, *The Java Programming Language,* Addison Wesley Publishing Company, Inc., 1996

[He97] Norman Hendrich, *Java für Fortgeschrittene,* Springer Verlag, Heidelberg, 1997

[St92] Bjarne Stroustrup, *Die C++ Programmiersprache,* Addison Wesley Publishing Company, Inc., 1992

[PP96] Ulla Kirch-Prinz, Peter Prinz, *C für PCs,* International Thomson Publishing, 1996

[He96] Eric Herrmann, *teach yourself CGI Programming with PERL in a week,* Sams.net Publishing, 1996

# 12 Anhang A

```
/*****************************************************************/
/*                                                               */
/*          LoadFSM.c -- CGI-Programm für JavaFSM                */
/*   von Ulrich Dallmann (2dallman@informatik.uni-hamburg.de)    */
/*   und Karola Krönert (2kroene@informatik.uni-hamburg.de)      */
/*                                                               */
/* Dieses Programm liest über stdin vom WWW-Server einen File-   */
/* namen und ein Passwort.                                       */
/* Im Filenamen werden unerlaubte Zeichen durch ein '_' ersetzt  */
/* und die Endung '.fsm' angehängt. Ist im spezifizierten Pfad   */
/* (siehe "path") ein solches File vorhanden, wird das darin     */
/* enthaltene Passwort überprüft und der neue Automat ggf. über  */
/* stdout gesendet. Ist noch kein File unter diesem Namen        */
/* vorhanden oder traten beim Lesen Probleme auf, so wird eine   */
/* HTTP konforme Fehlermeldung generiert.                        */
/*                                                               */
/*****************************************************************/

#include <stdio.h>
#include <stdlib.h>

main() {
  FILE *fp;
  char c;
  unsigned long i;
  unsigned long start,length;

  char work[101];                 /* zum Zwischenspeichern */
  char path[151] = "../fsm/";     /* um den Pfad zusammenzubauen */
  char name[101+4];               /* um den erhaltenen Namen zu merken */
  char passwd[101];               /* um das erhaltene Passwort zu merken
*/

  fgets(name, 101, stdin);        /* Zeile mit dem Dateinamen einlesen */
  fgets(passwd, 101, stdin);            /* Zeile mit dem Passwort einlesen
*/
  for (i=0; i<101; i++) {
    if (name[i]==10) name[i]=0;               /* <CR> ausfiltern */
    else if ( ((name[i]>0)&&(name[i]<48))||   /* alles bis zur '0',   */
              ((name[i]>90)&&(name[i]<97))||  /* zwischen 'Z' und 'a' */
               (name[i]>122) ) name[i]='_';   /* durch '_' ersetzen ! */
  }
  for (i=0; i<101; i++)
    if (passwd[i]==10) passwd[i]=0;     /* <CR> ausfiltern */
  strcat(path,name);                    /* Filenamen an Pfad hängen */
  strcat(path,".fsm");                  /* Endung noch anhängen */
  fp = fopen(path,"r");                 /* Datei zum Lesen öffnen */
  if (fp!=NULL) {                       /* wenn ok, dann ... */
    fgets(work, 100, fp);               /* erste Zeile einlesen */
    for (i=0; i<101; i++)
      if (work[i]==10) work[i]=0;       /* <CR> ausfiltern */
    if (strcmp(work,passwd)==0) {       /* Passwörter vergleichen */
      start = ftell(fp);                /* jetzige Lese-Position merken */
      fseek(fp,0L,2);                   /* zum Datei-Ende */
      length = ftell(fp) - start;       /* Länge des restlichen Files */
      fseek(fp,start,0);                /* zurück zur alten Position */

      printf("Content-type: text/plain\n");         /* HTTP Header */
      printf("Content-length: %ld\n\n",length+3);   /* HTTP Header */
      printf("OK\n");                   /* Rückmeldung: es folgen Daten */
      for (i=0; i<length; i++) {        /* restliche Zeichen senden: */
        c=fgetc(fp);                    /* Zeichen lesen */
        fputc(c,stdout);                /* und senden */
      }
    }
    else {                                  /* Paßwort ist falsch */
      printf("Content-type: text/plain\n"); /* HTTP Header */
      printf("Content-length: 18\n\n");     /* HTTP Header */
      printf("Passwort - Fehler\n");        /* Fehlermeldung */
```

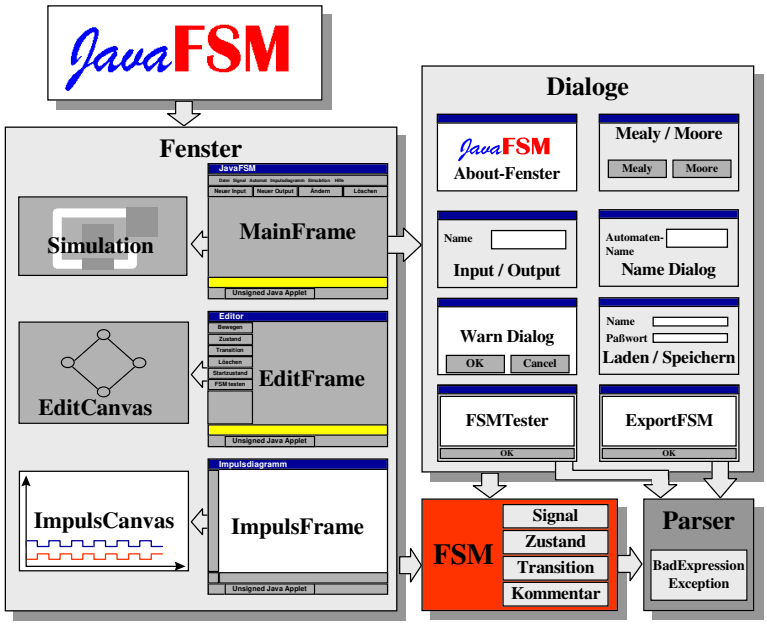
```
    }
    fclose(fp);                                /* Datei schließen */
  }
  else {
                                               /* konnte Datei nicht öffnen*/
    printf("Content-type: text/plain\n");      /* HTTP Header */
    printf("Content-length: 15\n\n");          /* HTTP Header */
    printf("Datei - Fehler\n");                /* Fehlermeldung */
  }
  return 0;
}
```

# 13 Anhang B

```
/*****************************************************************/
/*                                                               */
/*          SaveFSM.c -- CGI-Programm für JavaFSM                */
/*   von Ulrich Dallmann (2dallman@informatik.uni-hamburg.de)    */
/*   und Karola Krönert (2kroene@informatik.uni-hamburg.de)      */
/*                                                               */
/* Dieses Programm liest über stdin vom WWW-Server einen File-   */
/* namen, ein Passwort und eine Automatenbeschreibung.           */
/* Im Filenamen werden unerlaubte Zeichen durch ein '_' ersetzt  */
/* und die Endung '.fsm' angehängt. Ist im spezifizierten Pfad   */
/* (siehe "path") bereits ein File vorhanden, wird das darin     */
/* enthaltene Passwort überprüft und der neue Automat ggf. ge-   */
/* speichert. Ist noch kein File unter diesem Namen vorhanden,   */
/* wird der Automat direkt gespeichert.                          */
/* Im Erfolgsfall wird "OK", ansonsten eine Fehlermeldung        */
/* zurückgegeben. Beides geschieht HTTP konform.                 */
/*                                                               */
/*****************************************************************/

#include <stdio.h>
#include <stdlib.h>

main() {
  FILE *fp;
  char c;
  int i;
  long length;

  char work[101];                 /* zum Zwischenspeichern */
  char path[151] = "../fsm/";     /* um den Pfad zusammenzubauen */
  char name[101+4];               /* um den erhaltenen Namen zu merken */
  char passwd[101];               /* um das erhaltene Passwort zu merken
*/
  int valid = 0;

  const char *str_len=getenv("CONTENT_LENGTH"); /* Anzahl der zu lesenden
Bytes */
                                  /* ist in Umgebungsvariable */
  if (str_len!=NULL) {
    length=strtol(str_len,NULL,10);
    fgets(name,101, stdin);
    fgets(passwd,101, stdin);
    length -= (strlen(name)+strlen(passwd));
    for (i=0; i<101; i++) {
      if (name[i]==10) name[i]=0;
      else if ( ((name[i]>0)&&(name[i]<48))|| /* alles bis zur '0',   */
                ((name[i]>90)&&(name[i]<97))||/* zwischen 'Z' und 'a' */
                 (name[i]>122) ) name[i]='_'; /* durch '_' ersetzen ! */
    }
    for (i=0; i<101; i++)
      if (passwd[i]==10) passwd[i]=0;         /* <CR> ausfiltern */
    strcat(path,name);                        /* Filenamen an Pfad hängen
*/
    strcat(path,".fsm");                      /* Endung noch anhängen */
    fp = fopen(path,"r");                     /* Datei zum Lesen öffnen */
    if (fp!=NULL) {                           /* wenn ok, dann ... */
      fgets(work, 100, fp);                   /* erste Zeile einlesen */
      for (i=0; i<101; i++)
        if (work[i]==10) work[i]=0;           /* <CR> ausfiltern */
      if (strcmp(work,passwd)==0) valid=-1;   /* Passwort vergleichen */
      fclose(fp);                             /* Datei schließen */
    }
    else valid=-1;                        /* wenn Datei nicht vorhanden */

    if (valid==-1) {                      /* wenn soweit ok, dann */
      fp = fopen(path,"w");               /* zum schreiben öffnen */
      if (fp!=NULL) {                     /* wenn ok, dann ... */
        fprintf(fp,"%s\n",passwd);        /* Passwort zuerst speichern */
        for (i=0; i<length; i++) {        /* dann restliche Daten: */
```

```
          fscanf(stdin, "%c", &c);      /* ein Zeichen lesen */
          fprintf(fp, "%c", c);         /* und gleich speichern */
        }
        if (fclose(fp)!=EOF) {          /* Datei schließen & überprüfen */
          printf("Content-type: text/plain\n");     /* HTTP-Header */
          printf("Content-length: 3\n\n");          /* HTTP-Header */
          printf("OK\n");               /* alles erfolgreich verlaufen */
        }
        else {                          /* Speichern ist fehlgeschlagen */
          printf("Content-type: text/plain\n");     /* HTTP-Header */
          printf("Content-length: 17\n\n");         /* HTTP-Header */
          printf("Schreib - Fehler\n");             /* Fehlermeldung */
        }
      }
      else {                            /* konnte Datei nicht öffnen */
        printf("Content-type: text/plain\n");       /* HTTP-Header */
        printf("Content-length: 15\n\n");           /* HTTP-Header */
        printf("Datei - Fehler\n");                 /* Fehlermeldung */
      }
    }
    else {                                      /* Paßwort ist falsch */
      printf("Content-type: text/plain\n");         /* HTTP-Header */
      printf("Content-length: 36\n\n");             /* HTTP-Header */
      printf("Datei vorhanden / falsches Passwort\n"); /* Fehlermeldung */
    }
  }
  else {                                        /* keine Umgebungsvariable
*/
    printf("Content-type: text/plain\n");           /* HTTP-Header */
    printf("Content-length: 43\n\n");               /* HTTP-Header */
    printf("CGI - Fehler\n");                       /* Fehlermeldung */
    printf("CONTENT_LENGTH nicht gesetzt!\n");      /* noch Fehlermeldung
*/
  }
  return 0;
}
```

# 14 Anhang C: package JavaFSM




Class Index

| Klasse | Beschreibung |
|---|---|
| JavaFSM | Die Hauptklasse (Applet und Applikation) |
| FSM | Automaten-Objekt, mit zugehörigen Methoden |
| MainFrame | Das Hauptfenster |
| Simulation | Das Panel mit dem Schaltbild (im Hauptfenster) |
| ImpulsFrame | Das Fenster mit dem Inpulsdiagramm |
| ImpulsCanvas | Das Canvas, in dem das Inpulsdiagramm gezeichnet wird |
| EditFrame | Das Fenster mit dem Automaten-Editor |
| EditCanvas | Der Canvas, in dem der Automat entworfen wird |
| Kommentar | Kommentar-Objekt |
| Parser | Der Parser für logische Funktionen |
| FSMTester | Dialogfenster zum Überprüfen eines Automaten |
| SignalDialog | Dialogfenster zum Einfügen von Signalen |
| Signal | Signal-Objekt (Input / Output) |
| Zustand | Zustand-Objekt |
| Transition | Transition-Objekt |
| MMDialog | Dialog zur Auswahl des Machine-Types (Mealy / Moore) |
| NameDialog | Dialog zum Ändern des Maschinen-Namens |
| UserDialog | Dialog zum Laden / Speichern von Automaten |
| ExportFSM | Dialog zum Exportieren des Automaten nach VHDL / KISS |
| Statuszeile | Statuszeile in Hauptfenster und Editor (gelb) |
| Panel3D | einfaches Panel mit 3D-Umrandung |
| WarnDialog | Dialogfenster, warnt den Benutzer vor Datenverlust beim Wechsel in den Editiermodus |
| About | Info-Fenster mit Versionsnummer |

# Exception Index

| Klasse | Beschreibung |
| --- | --- |
| BadExpressionException | Parser-Exception für fehlerhafte Ausdrücke |