

THE TECHNICAL UNIVERSITY OF ŁÓDŹ
Faculty of Electrical and Electronic Engineering

Master of Engineering Thesis

Visualization of Systems Applied to FSM Control
Piotr Cieciera

Student's number: 106107

Supervisor:
dr inż. Grzegorz Jabłoński

Auxiliary supervisor:
mgr inż. Bogusław Kosęda
mgr inż. Wojciech Cichalewski

Łódź, 2006

1 Wstęp

Głównym celem pracy magisterskiej było stworzenie programu, wykorzystującego graficzny interfejs użytkownika (ang. GUI), do monitorowania pracy maszyny stanowej (ang. FSM) i ukazującego jej wewnętrzną strukturę.

Praca została stworzona w ramach projektu CARE (Coordinated Accelerator Research in Europe). Program ma współpracować z maszynami stanowymi kontrolującymi pracę akceleratora liniowego w Deutsches Elektronen Synchrotron (DESY) w Hamburgu [5].

W czasie projektowania i obsługi maszyny stanowej potrzebne jest jej zdalne monitorowanie dla celów testowych i diagnostycznych w formie systemu wizualizacji. System taki powinien w szczególności posiadać intuicyjny interfejs, powinien być niezależny od plików konfiguracyjnych maszyny, powinna istnieć możliwość uruchomienia go z przeglądarki internetowej.

Rozwiązanie powinno być wygodne dla użytkownika, wieloplatformowe i osiągalne z dowolnego komputera sieci lokalnej, dlatego jako język programowania wybrana została Java i technologia apletów. Dodatkowo Java SE ma wbudowane biblioteki graficzne umożliwiające tworzenie GUI.

W rozwiązaniu wykorzystano architekturę klient – serwer. Stworzony program jest klientem a maszyna stanowa – serwerem. Klient żąda dwóch typów danych dostępnych na serwerze. Pierwszym jest struktura FSM w formie XML – są to dane statyczne, drugim – dynamicznie uaktualniana informacja o obecnym stanie maszyny.

Struktura FSM jest zapisana w postaci pliku XML i przechowywana na serwerze maszyny stanowej jako jej parametr. Maszyny stanowe tworzone są w środowisku MATLAB 7.0 rozszerzonym o Stateflow Toolbox, dlatego zaistniała konieczność dodania nowej funkcjonalności tzn. możliwości tworzenia plików XML bezpośrednio ze Stateflow.

Graficzna reprezentacja FSM została zaimplementowana w postaci grafu. Początkowo powstała biblioteka umożliwiająca tworzenie bloków i strzałek grafu, ale wymagania co do funkcjonalności wymusiły użycie profesjonalnej biblioteki typu Open Source. Biblioteka ta posiada możliwość wizualizacji grafu.

Wszystkie założenia zostały zrealizowane tak aby uzyskać jak najlepszą funkcjonalność projektu przy dostępnych narzędziach programistycznych.

2 Abstract

The main goal of this master thesis was to develop a computer program utilizing Graphical User Interface to monitor work of Finite State Machine and showing its internal structure.

This work is strongly connected with Coordinated Accelerator Research in Europe (CARE) project. This program will cooperate with FSMs controlling work of accelerator at Deutsches Elektronen Synchrotron (DESY) in Hamburg. [5].

The task was undertaken because there was a need for online monitoring of a FSM for test and diagnostic purposes in a form of visualization system, that in particular should have an intuitive interface, shouldn't rely on FSM's configuration files and it should be possible to execute it from the Internet browser.

The solution had to be comfortable for an end user, multiplatform and reachable from any computer in a local network, therefore Java Standard Edition was chosen as a programming language together with the applet technology. In addition Java SE comes with graphical libraries specified for GUI development.

This solution utilizes client – server architecture in which this program is a client and FSM is a server. Client requests two types of data which are available on a server: first is a structure of a machine in form of XML – these are static data, second is a dynamically updated information about the current state of a machine.

Structure of FSM is described in form of XML file and stored on the machine server as its property. FSMs are created in MATLAB 7.0 environment with the Stateflow toolbox therefore additional functionality was added, which is, a creation of a XML file from Stateflow.

Implementation of visual representation of FSM in the program is in a form of a graph. At the beginning new library was written for creation of blocks and arrows of a graph but the requirements of functionality force utilization of a professional open source library.

This library comes with an extension which helps create planar layout of a given graph. All the main requirements are fulfilled in a way that balances the functionality of an applet with the availability of programming tools.

3 Contents

1	WSTEP.....	1
2	ABSTRACT.....	3
3	CONTENTS.....	5
4	INTRODUCTION.....	7
4.1	<i>RATIONALE AND THE PROBLEM DEFINITION.....</i>	12
5	FINITE STATE MACHINE.....	14
5.1	<i>THEORETICAL INTRODUCTION.....</i>	14
5.2	<i>THE STATEFLOW TOOLBOX.....</i>	17
5.2.1	<i>Stateflow syntax.....</i>	17
5.2.2	<i>Stateflow semantics.....</i>	21
5.3	<i>FSM APPLICATION.....</i>	23
5.3.1	<i>General.....</i>	23
5.3.2	<i>In DESY center.....</i>	24
6	VISUALIZATION OF SYSTEMS.....	25
6.1	<i>USER INTERFACE.....</i>	25
6.2	<i>GRAPHICAL USER INTERFACE.....</i>	25
6.3	<i>VISUALIZATION METHODS FOR FSM.....</i>	28
7	PROBLEM SOLUTION – STRUCTURE.....	32
7.1	<i>REQUIREMENTS.....</i>	32
7.2	<i>CLIENT-SERVER MODEL.....</i>	35
7.2.1	<i>Server part.....</i>	35
7.2.2	<i>Client part.....</i>	37
8.	PROBLEM SOLUTION - IMPLEMENTATION.....	40
8.1	<i>MATLAB STATEFLOW PART.....</i>	40
8.1.1	<i>Data structure in XML.....</i>	40
8.1.2	<i>MATLAB functions.....</i>	43
8.2	<i>APPLET PART.....</i>	50
8.2.1	<i>Applet framework.....</i>	50
8.2.2	<i>Graphical User Interface.....</i>	51
8.2.3	<i>Visualization Area.....</i>	54
9	USER MANUAL.....	58
9.1	<i>INSTALLATION INSTRUCTIONS.....</i>	58
9.2	<i>USER OPERATIONS.....</i>	58
9.2.1	<i>Running the program.....</i>	58
9.2.2	<i>User Interaction.....</i>	58

10	CONCLUSIONS AND SUMMARY	64
11	REFERENCES.....	67
12	APPENDIX.....	69

4 Introduction

Project of visualization of systems applied to FSM control is connected with the Coordinated Accelerator Research in Europe (CARE) project. This program includes the most advanced scientific and technological developments relevant to accelerator research for particle physics. Its aim is to foster and strengthen the European knowledge to evaluate and develop efficient and cost effective methods to produce intense and high-energy electron, proton, muon and neutrino beams as recommended by the European Committee for Future Accelerator (ECFA) [1].

The participants will integrate their infrastructures, establishing a European technological platform for accelerator research.

The research activities include the developments of:

- Superconducting cavity and RF technology
- Photo-injector technology, in particular for two-beam acceleration technique
- Normal and superconducting structures for the acceleration of very high-intensity proton beams as well as challenging beam chopping magnets
- The technology for constructing very high magnetic field and high density currents magnets.

Twenty two contracting participants and a large number of associated institutes participate in this integrating effort. The CARE project represents an innovative and unique opportunity in Europe as it will involve almost all of the European expertise and know-how in accelerator physics and related technologies and would allow one to address most of the issues relevant to particle accelerators. Thus, it will provide an integrated service to the entire European particle physics community and could provide on the long-term an integrated service for other communities as well.

One of the participants is the **Deutsches Elektronen Synchrotron (DESY)** research center in Hamburg, Germany [2].

It contributes mainly in the field of:

- Large Scale Facilities (accelerator complex – PETRA, HERA, ILC)
- Large-scale accelerator test facilities (TESLA Test Facility, X-FEL, Horizontal Cryogenic test stand)

ILC (International Linear Collider) comprises a 33-kilometer-long superconducting linear accelerator, which will bring electrons into collision with their antiparticles, the positrons. This high-energy particle collisions will allow physicists to take a closer, more detailed look than ever before at the structure and origin of matter and the universe.

With a linear collider like ILC, particle physicists can study what happened just after the big bang. New superconducting acceleration structures, so-called cavities (resonators), bring electrons and positrons to record energies of 250 to around 500 billion electronvolts each. The particles collide head-on in the middle of the 33-kilometer-long TESLA collider. These energies are on a par with those present during the first 10^{-12} of a second of the universe's existence. Particles and antiparticles annihilate each other to create a tiny "fireball" of concentrated energy. As was the case in the big bang, an extremely broad variety of elementary particles are created from this energy, including, the physicists are hoping, the much sought after "Higgs" and "SUSY" particles.

It is the use of superconducting technology that distinguishes TESLA from linear collider designs using normally conducting resonators. TESLA's acceleration sections make it possible to create a particle beam of optimum quality with high collision rate of the accelerated particles, which is the ideal prerequisite for new discoveries in particle physics.

The **TESLA Test Facility** in Hamburg consisted of an electron source, a superconducting acceleration section and an arrangement of magnets in which the tests on the accelerated electrons were conducted to produce flashes of X-ray laser light (free-electron laser).

The research successes achieved at the test facility:

- superconducting accelerator modules have been developed in cooperation with industrial companies;
- a world first - short-wavelength ultraviolet laser beams have already been produced using the new method foreseen for the X-ray laser;
- experiments carried out using the free-electron laser (FEL) - the interaction of matter with X-ray radiation from an FEL on extremely short time scales for the first time.

The **X-ray free-electron laser (XFEL)** that is being planned at the DESY research center in cooperation with European partners will produce high-intensity ultra-short X-ray flashes with the properties of laser light. Thus scientists soon will be able to film events in the microcosm and find out how materials and biomolecules behave at the atomic level, how a chemical reaction progresses, how biomolecules move, and how solids are formed. This will benefit a wide range of natural sciences - from physics and chemistry to materials science, geological research and the life sciences. It could also offer very promising opportunities for industrial users.

The new free-electron laser **VUV-FEL** - the pilot facility for the XFEL, which generates vacuum ultraviolet (VUV) and soft X-ray radiation in a range down to wavelengths of six nanometers, was commissioned in 2004, making possible groundbreaking experiments. TESLA test facility was modified to a total length of 260 meters and extended to the VUV-FEL. The new free-electron laser consists of a series of sequentially connected superconducting resonators that accelerate the electron beam to an energy of 1 billion electron volts, which generates soft X-ray radiation. The intense light flashes are then distributed among a total of five measuring stations. It is also the pilot facility for the XFEL, because its operation will provide major insights that will benefit the X-ray laser XFEL, which will generate light having even shorter wavelengths, down to one-tenth of a nanometer.

Such a complex facility as VUV-FEL, the current step on the way to develop a linear accelerator, needs a reliable and efficient control system. To test cavities, to diagnose and to perform numerous measurements the system has to be flexible. As a machine for user experiments a very reliable and stable operation is required. Since this accelerator is a short prototype for a future XFEL and a linear collider also, the design of the control system has to be scalable and take these requirements into account as well. That is a task of **Distributed Object Oriented Control System (DOOCS)**. The design of DOOCS is based on device objects that represent the real world hardware devices. This leads to a concept of device servers which handle all properties of a particular device and a single software unit that controls a complete device instance. Devices are then created with the powerful methods of object oriented languages. The same object oriented design concepts are also used in the network communication and in client programs [3].

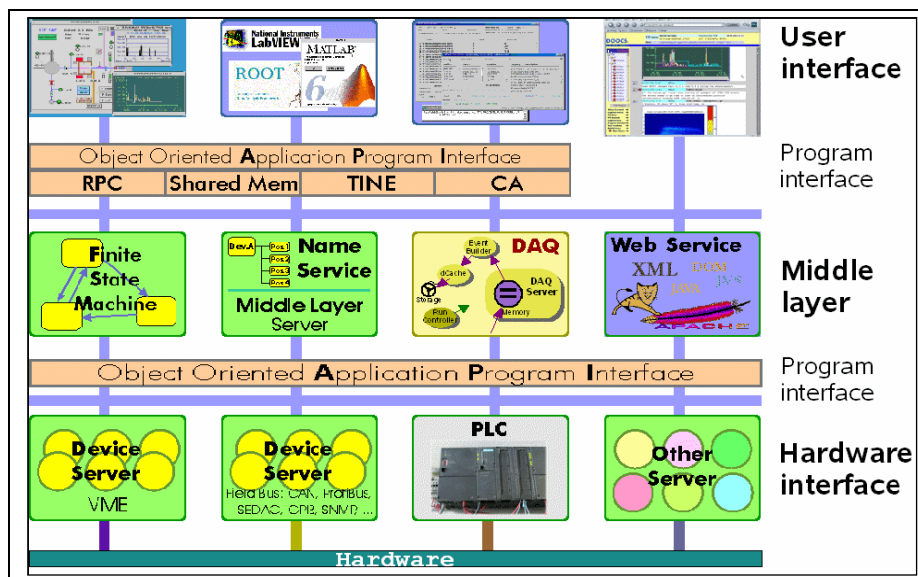


Figure 1.1 DOOCS architecture

The DOOCS architecture is based on three main layers: (Fig. 1.1) [4]

- Application programs with a user interface are in the top layer. All these programs connect to the other two layers via a common API.
- The middle layer contains higher level services.
- On the bottom layer all services with device connections are located.

Considering the DOOCS architecture one come to the main subject of this thesis which is the **Finite State Machine** and its visualization.

Finite State Machine server is a part of system, which among other services (Name Server, Data Acquisition Server, Web services), is implemented in the middle layer of DOOCS.

TESLA Test Facility involves people from all over the world operating the accelerator. As not all of them can be experts in TTF operation, a high degree of automation is necessary to relieve the operators from complex but well understood tasks. Besides, TTF will become a user facility for VUV-FEL experiments and therefore will have to provide stable modes of operation. This requires a highly automated machine because of the complexity involved.

A common approach to automation is the use of Finite State Machines. It is a well understood approach that is used in various industrial control systems. This resulted in the idea of implementing a high level FSM with the goal of the “One-Button Operation”.

4.1 Rationale and the problem definition

Development phase of any FSM requires many iterations of tests and upgrading procedures. A very advanced and flexible software environment of MATLAB with Stateflow Toolbox is used in the stage of designing an FSM. Intuitive Graphical User Interface of this software speeds up designer work and makes it comfortable even in a case of highly advanced machine with hundreds of internal states. That is provided through such features as: coping, pasting of structures and automatic deletion of cyclic dependencies. Tests of such a machine are also performed on-the-fly with syntax being checked automatically.

Tests in real or test environment are another step in a development process. After generation of C++ code Finite State Machine has to perform its work connected to the real devices, manage real signals being sent or received. Unfortunately FSM control on a given system can only be seen as the resulting changes in the system properties. Finally the measurements from the dedicated sensors and its comparison to the correct values, gives designer the ability to draw the conclusions about the correctness of an FSM. As one may notice it is not reliable procedure, some readouts may change rapidly, some may not be even noticed. The amount of readouts and its translation to the FSM behaviour requires a lot of work from tester and in-depth knowledge of a controlled system.

Therefore there is a need for a tool that would:

- show the internal structure of an FSM;
- be able to provide an information about its current state, the state that it was changed from and parameters that triggered this transition;
- be independent from any configuration files or source code of an FSM.
- record history of changes

Such a monitoring ability would be very helpful for testing purposes. In contrary with the present form of testing this would decrease time spent on testing.

To improve efficiency of this tool, it should have the Graphical User Interface which gives the advantage of:

- symbols being recognized faster than text,
- faster usage and problem solving,
- increased feeling of control.

Very important part of such a tool would be the graphical representation of a Finite State Machine because interaction with it is the most intuitive and most comfortable for the end-user.

The issue of efficiency and interface being intuitive is also important for the operator of the control system. The one who will be responsible for the operation of the system, could faster react in case of problems and minimize the break-time because visualization gives better understanding of the current status of FSM.

As far as the infrastructure of the system is considered such a solution should be:

- available from any computer in the network
- multiplatform
- available from the most common interface of a Web Browser.

5 **Finite State Machine**

Visualization of systems is the main subject that is covered by this thesis. A Finite State Machine, also called Finite Automaton, is the object of this visualization. Therefore there is a need to describe this concept. A formal definition is usually the most coherent, so it will be presented first. Some implementations of them in real-live systems are then described, especially in Distributed Object Oriented Control System in DESY.

5.1 *Theoretical introduction*

Studying the literature on state automata one for sure meets general description of FSM as a model of computation composed of states, start state, transition function and input alphabet.

- A state stores information about the past.
- A transition function indicates a state change.
- An input alphabet consists of all possible inputs to the machine.
- A start state is the condition of the machine in which computation begins.

There exist a classification of FSMs on acceptor FSMs and transducer FSMs together with its mathematical models. Following [6] and [7] a Finite State Machine is defined as:

1. In an Acceptor form as a quintuple $\langle S, I, \delta, s_0, F \rangle$, where:
 - S is a finite non empty set of states.
 - I is the input alphabet (a finite non empty set of symbols).
 - $\delta: S \times I \rightarrow S$ is the state transition function.
 - s_0 is an initial state, an element of S .
 - F is the set of final states.

2. In a Transducer form as a sextuple $\langle S, I, B, s_0, \delta, \omega \rangle$, where:

- S is a finite non empty set of states.
- I is a finite non empty set of inputs.
- B is a finite non empty set of outputs.
- s_0 is the initial state, an element of S .
- $\delta: S \times I \rightarrow S$ is the state transition function.
- ω is the output function.

If ω is a function of a state and inputs ($\omega: S \times I \rightarrow B$) that definition corresponds to the Mealy model.

If the output function depends only on a state ($\omega: S \rightarrow B$) that definition corresponds to the Moore model.

To understand those formal definitions a more intuitive description is needed. State reflects the input changes from the system start to the present moment. Computation begins with a start state. State change indicated by transition function is described by a condition that would need to be fulfilled to enable the transition. An action (output) defines an activity that is to be performed at a given moment.

There are several action types:

- entry action - executed when entering the state
- exit action - executed when exiting the state
- input action – executed dependent on present state and input conditions
- transition action - executed when performing a certain transition

Let's look at the classification previously stated.

A binary output from the **acceptor** machine is either yes or no and defines whether the input is accepted by the machine or not. All states of the FSM are said to be either accepting or not accepting. If all input is processed and the current state is an accepting state, the input is accepted otherwise not. The machine is often used as defining a language. It accepts or rejects input words.

Transducer machine generate output based on a given input and/or a state. It is used for control applications. Here two types are distinguished: [8]

- Moore machine

In this kind of FSM output depends only on the state. The advantage of the Moore model is a simplification of the behavior.

- Mealy machine

In this kind of FSM output depends on input and state. The use of a Mealy FSM leads often to a reduction of the number of states as compared to Moore machine.

A further distinction is between deterministic (DFA) and non-deterministic (NDFAs, GNFA) automata [9]. Deterministic automata for each state has exactly one transition for each possible input. Non-deterministic one has none or more than one transition from a given state for a given possible input.

5.2 The Stateflow Toolbox

To better understand Finite State Machine's functionality and behavior it is often presented as a state diagram. This is a graphical representation which utilizes basic symbols to show the FSM configuration at a glance.

One of the graphical design and development tools, which incorporates and even extends the concept of a state diagram, is the Stateflow Toolbox from the MATLAB environment. It is used to simulate complex reactive systems based on FSM theory [10].

Stateflow gives the possibility to design and develop deterministic, supervisory control systems in a graphical environment. It visually models and simulates complex reactive control to provide clear, concise descriptions of complex system behavior using finite state machine theory, flow diagram notations, and state-transition diagrams all in the same diagram. Stateflow brings system specification and design closer together. It is easy to create designs, consider various scenarios, and iterate until the Stateflow diagram models the desired behavior. Thus, it is very helpful to understand Stateflow syntax and semantics.

5.2.1 Stateflow syntax

The behavior of an event-driven system is modeled by describing it in terms of transitions among states. Stateflow extends this notation by a variant of the finite state machine, namely Statechart, established by Harel. This work is described in [11].

Statecharts in Stateflow are called diagrams and are a graphical representation of a finite state machine, with states and transitions as the basic building blocks of the system.

STATES

Finite state machines model objects that have a limited number of states (Fig. 5.1). A state can be active or inactive.

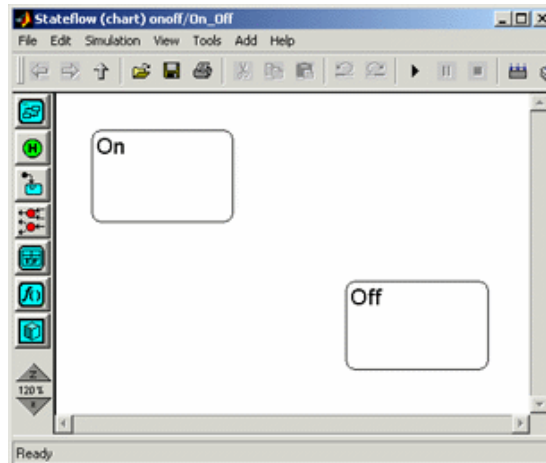


Figure 5.1 States in Stateflow

TRANSITIONS

A relation that defines an order of change of active states is called transition. Transitions are directional what means that they start at a source state and end in a destination state. If the source state is active, after the transition, the source state becomes inactive and the destination state becomes active. A default transition, shown in Fig. 5.2, indicates the initial state.

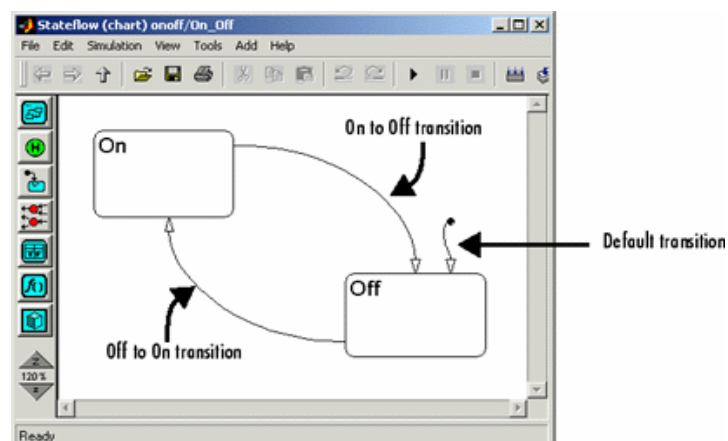


Figure 5.2 Types of transitions in Stateflow

FLAWS

Stateflow provides two types of states: with exclusive (OR) and with parallel (AND) decomposition. Exclusive (OR) states are used to describe modes that are mutually exclusive (only one state can be active at a time). In contrary parallel (AND) states called flows can be active at the same time. (Fig. 5.3)

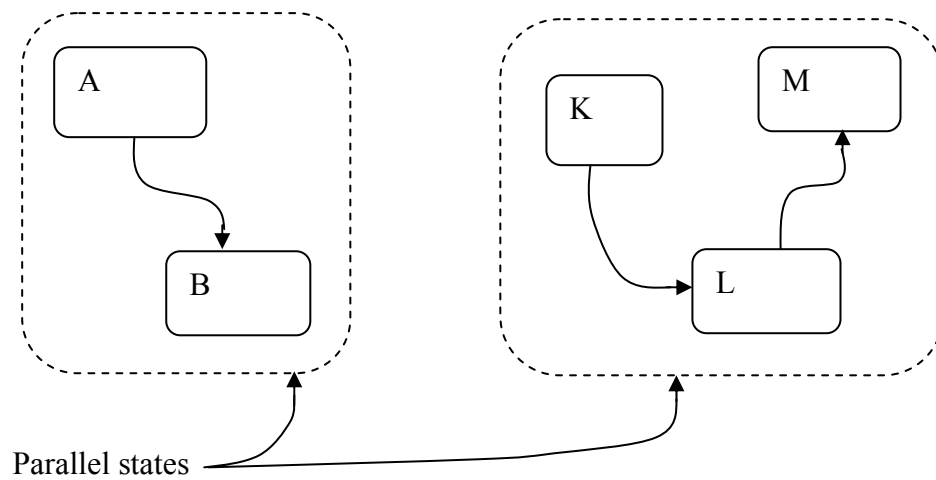


Figure 5.3 Flows (parallel states) in Stateflow

CONDITIONS

A condition is a Boolean expression which specifies that a transition occurs if the specified expression is true. Denoted in square brackets in Fig. 5.4.

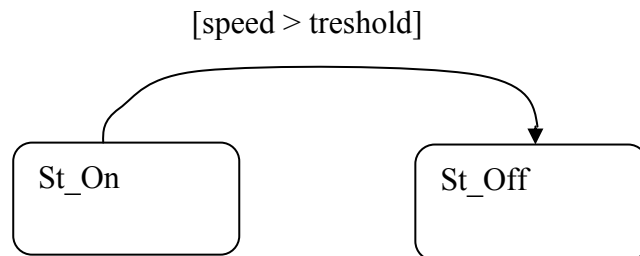


Figure 5.4 Use of conditions in Stateflow

EVENTS & ACTIONS

Events and actions are not graphical objects but they are the things that make that the statechart works. Events (input symbols) trigger the transitions between states. Actions are the outputs of system which take place while Stateflow diagram is executed. They happen as a part of transition execution or state change.

Transition label is the place where action is defined:

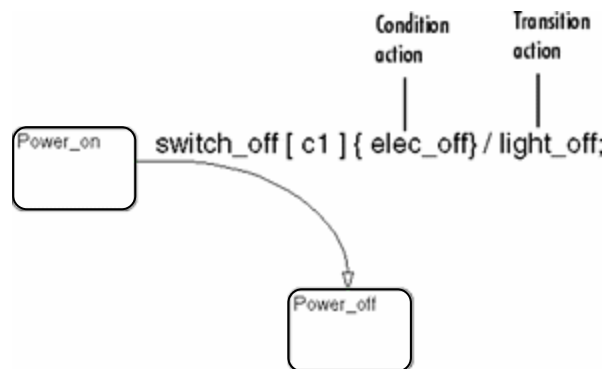


Figure 5.5 Transition label format

In state label one can define different types of actions:

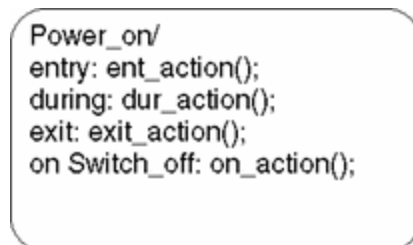


Figure 5.6 Actions in state label

Additionally, Stateflow enables the representation of hierarchy, history and parallelism described above. Hierarchy gives possibility to organize complex systems by defining a parent - children object structure. History provides the means to specify the destination state of a transition based on historical information.

5.2.2 Stateflow semantics

Operational semantics describe how the objects described in the previous paragraph are interpreted and implemented and in what sequence actions take place during Stateflow diagram execution.

The following statements are primary axioms of proper Stateflow behavior:

- Whenever a state is active, its parent should also be active.
- A state (or chart) with exclusive (OR) decomposition must never have more than one active child.
- If a parallel state is active, siblings with higher priority (higher graphical position in the Stateflow diagram) must also be active.

EVENTS EXECUTION

A Stateflow chart executes only in response to an event. This occurs on two levels. First, Simulink updates the chart, which awakens it for execution. Second, once the chart is awakened, it continues to respond to events until there are no more events. The chart then goes to sleep. When another event occurs, the chart is awakened (from sleep) to respond to the event. All activity caused by the event in the chart is completed before returning to whatever activity was taking place prior to reception of the event.

TRANSITION EXECUTION

If a chart has exclusive (OR) states, its execution begins with the default transitions that point to the first active states in a chart. Any actions associated with the sources or destinations are related to the transition that joins them.

STATES EXECUTION

A state performs its entry action (if specified) when it becomes active. The state is marked active before its entry action is executed and completed.

Active states that receive an event that does not result in an exit from that state execute a during action to completion if a during action is specified for that state. An on event_name action executes to completion when the event specified, event_name, occurs and that state is active.

A state performs its exit action (if specified) before it becomes inactive. The state is marked inactive after the exit action has executed and completed.

5.3 FSM application

5.3.1 General

Finite state machines have its implementation in a hardware or software form. They generally model systems which can proceed in clearly separate and discrete steps from one to another of a finite number of configurations or states [12].

Some concepts created for industry purposes can be easily implemented in a digital circuit form using programmable logic device, a programmable logic controller, logic gates and flip flops. More precisely, a hardware implementation requires a register to store state variables, a block of combinational logic which determines the state transition, and a second block of combinational logic that determines the output of a FSM.

In addition to their use in modeling reactive systems, finite state automata are important in many other areas, including linguistics, computer science, logic, biology and mathematics.

They marked its strong position mainly in fields related to computer science e.g.:

- Bioinformatics
- Compilers
- Computer-aided verification
- Data and image compression
- Design and architecture of software
- DNA/molecular computing
- Document engineering
- Natural language processing
- Networking
- Object-oriented modeling
- Pattern-matching
- Speech and speaker recognition
- Spell checking
- Symbolic manipulation
- Text processing
- VLSI
- World-wide web

5.3.2 In DESY center

Finite State Machine has already its place in the whole Distributed Object Oriented Control System as a service between the low-level (hardware) layer and high-level (operator) layer of the system. The designed and developed FSMs are the source of automation of the system. The tendency to realize the goal of “One-Button-Operation” would relieve the operators from complex but well understood tasks providing increased level of safety as human actions are often considered as error prone [13].

Nowadays, FSM designers work on a development of state automata for two main parts of VUV-FEL accelerator, namely: RF-power station and Low Level RF control system. Final realisation of FSMs will give the following advantages: [14]

- Protection of klystron’s hardware against damage – overloading avoidance
- Provide maximum lifetime of klystron and modulator
- Maximize availability of klystron and modulator for accelerator operation
- Components standardization
- Remote diagnostic

FSM is planned to automate the following functions and procedures:

- Turn on, turn off and reset klystron and all subsystems
- Check whether all subsystems are ready
- Set proper parameters of those subsystems
- Manage interlocks

The design and test phase on a software level is performed in the previously described Stateflow Toolbox. Another advantage of this environment is a creation of the FSM as a computer program in a C language, what makes the implementation phase much easier.

6 Visualization of Systems

This chapter considers main topic of the thesis which is visualization of systems. **Visualization** of a particular system is a presentation of a meaningful information contained in this system by means of graphical objects.

Its goal is to easily understand:

- function of the system
- relations between parts of the system
- interaction of the system with its environment

Applying some interaction to the graphical objects, extends this presentation to a form of User Interface.

6.1 User Interface

This consideration treats user interface design as a subset of a field of study called human-computer interaction (HCI) which studies how people and computers work together that user needs are satisfied.

User interface is a collection of techniques and mechanisms used to interact with a computer system. It may be: text, voice, handwriting, pointing, pictures, video, etc. [15].

6.2 Graphical User Interface

In a graphical user interface the main interaction mechanism is pointing. It comes from movements and gestures which are on the lowest and most common level of communication.

This kind of interface utilizes graphical symbols which correspond to the real objects or behaviors. Graphical objects are always visible and have a set of common actions assigned. They are used to perform some tasks by direct manipulation of them.

The idea of connection of graphics and manipulation constitutes present graphical interfaces. What makes graphics so popular? A graphical screen switches from plain text-based interfaces to the three-dimensional look. All symbols are assumed to imitate 3D objects. The user screen gain depth by such functionality as: floating windows, etched borders, controls that rise above the screen, objects that overlap each other, pull-down menus. That creates a feeling of dealing with an ordinary desktop that all users are used to. One can find documents, notes, files, folders and recycle bin on it. The GUI becomes a familiar environment for end-users in which they are able to perform its work quickly and dynamically.

Using graphical presentation method user capabilities are utilized more effectively. It requires less information recording, memory is not overloaded because of a compact representation of information. Simple, but time-consuming procedures are represented by visual symbols – icons. Presentation of large amounts of data in a graphical form invokes faster comparisons of amounts, trends, or relationships.

Another important thing and very useful for an end-user is a customization of its environment what increase user connection with its place of work or leisure.

Direct manipulation method is also worth describing. Important thing is that users work in an environment that imitates the real one. They are able to perform almost all actions that usually do in a intuitive way, the implementation of this actions is hidden. That gives a possibility to focus on a data not the tool that has to be used.

All objects are constantly visible what makes users conscious what they are doing. This concept is called: WYSIWYG - what you see is what you get.

Actions are rapid with visible display of results and progress what provides feedback for an user. Actions which are incorrect can be easily undone.

Synergy of the described techniques of graphical visualization and direct manipulation is easily seen in everyday live. Its main advantage is that ordinary users hardly ever notice the amount of work that is done behind the scenes to provide them with such a powerful tool. But to utilize, extend or implement all functions that a GUI give, one has to be conscious of its advantages and disadvantages.

Starting with advantages:

- Graphical symbols are recognized faster then text – Special design of icons make them easier to classify.
- Easier remembering and faster learning
- Provide context and concrete thinking
- Increased feeling of control- because of immediate feedback
- System responses are predictable
- One can use it with less anxiety, actions are reversible
- Is more attractive and appealing, requires less typing
- Consumes less space – because of icons used

But there are also such a features which are considered as disadvantages:

- Creation of good GUI requires complex design and test process
- Some abstract phenomena are hard to visualize – it's easier to express them in a text form

6.3 Visualization methods for FSM

There are different forms of representing Finite State Machines. Except the mathematical definition presented previously, there exists a matrix description of a particular automata called transition matrix (Fig. 6.1).

	destination states				
	StA	StB	StC	StD	source states
StA	0	0	1	1	
StB	0	0	1	0	
StC	0	1	0	1	
StD	1	0	0	0	

Figure 6.1 FMS in a matrix form.

Matrix defines transitions that are present between states. State names describe rows and column of matrix. One have to assume row or column as a source or destination state. Transitions are coded with the boolean logic: value 1 – there exists a transition ,0 – there not exists.

This form of representation of FSM is an easy way to code it for computer calculations and algorithms which require strict knowledge and definitions.

The way which gives most perceptual and cognitive possibilities for human beings is a graphical representation. The most popular form is **state diagram** or in its extended form a statechart (used in Stateflow Toolbox, described in previous chapter).

State diagram is a form of a directed graph in which particular symbols and objects have its special meaning. As any graph, it consists of nodes and edges. Nodes are considered as states while edges represent transitions. The symbol for edge is actualy an arrow

defining its directivity (source and destination states). Following figure is an example of state diagram which is based on the same FSM defined in the matrix form.

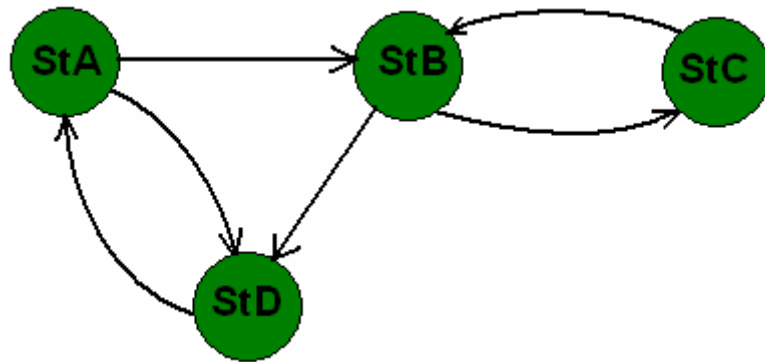


Figure 6.2 State diagram example

The following reasoning gives the answer why diagrams are best for visualization of finite state automata.

Representation in a form of a diagram utilizes so called conventional and perceptual symbols [15].

Conventional symbols are things that have some particular meaning because people agreed on this meaning. It is a matter of convention, something that rise from sociality, culture. Such symbols are usually used by specialists that construct some convention useful in describing some problems. Examples of that symbols are text labels. Text is read and understood because people agree on meaning of letters, further more acronyms are understand because people have some knowledge from a particular area. Others need additional explanation, for ex.: StA, StB ,... - states of the finite state machine. Conventional symbolics is also used to narrow the amount of possible interpretations of a presentation (arrow represent transition not the direction of movement, etc.).

Meaning of the label is strongly amplified by the use of perceptual symbolics. Perceptual symbols are commonly understood without learning its special meaning. They mostly comes from human psychology and cognitive abilities.

Finite state machines in a form presented in a Figure 6.2 utilizes this human abilities providing best understanding of the concept of FSM.

Diagrams are generally structures that express entity – relationship model with the following rules:

1. A closed contour in a node–link diagram generally represents an entity of some kind. In this case a state.
2. The shape of the closed contour is frequently used to represent an entity type – possible different types of states.
3. The color of an enclosed region represents an entity type – another form of differentiating states.
4. The size of an enclosed region can be used to represent the magnitude of an entity - the same size provides similarity.
5. A line linking entities represents some kind of relationship between them – transition between states. Arrows give this relation stronger meaning.
6. A contour can be shaped with tabs and sockets to indicate which components have particular relationship – additional extentions of the diagram are possible.

As one can notice, all principles that governs FSM work are satisfied using state diagram form of visualization. That comes from the very basic rules that human brain follows.

They are called ‘Gestalt Laws’ (in German: pattern) and here is their short description:

[16]

- Spatial proximity – things that are close together are considered as grouped.
- Similarity - similar pattern of elements determine their grouping.
- Connectedness – the most powerful principle that determines groups of objects

___ and relation between them.

- Continuity – smooth and continuous elements are better for visualization of entities, abrupt change in directions cause problems in recognizing of the meaning.
- Symmetry – provides powerful organizing principle.
- Closure – closed contour tend to be seen as objects, human brain tends to close contours and differentiate the inside and outside of space.
- Relative size – smaller components are seen as objects, instead of large spaces, also provides the similarity feature.
- Figure and ground – effect that helps identify objects from the background.

7 Problem Solution – Structure

7.1 Requirements

The main objective that had to be complete by the project is the visualization of the structure of an FSM and its current state.

The requirements that are specified for the project are as follows:

- a) solution available from any computer in a network
- b) multiplatform
- c) utilizes Graphical User Interface
- d) visualization in an intuitive an non-problematic form

Ad. a)

To realize the first requirement, the computer environment in DESY had to be explored, especially computer platforms types together with a network availability have to be known.

Generally, the desktop workstations are configured as:

- a PC or a SUN hardware
- with Windows XP Professional, Linux Suse or Sun Solaris operating systems

Workstations are connected using Ethernet 10/100 Mbit/s network.

Ad. b)

The next requirement, that the solution has to be multiplatform, is strongly connected with the first one. It has to be solved to provide the availability of the application.

Development of the software that is going to operate on different platforms needs special consideration.

- It can be written in a language for which compilers for dedicated platforms exists. In such a case one source code is created, then it is compiled for each destination platform independently.
- Alternative solution is to take the advantage of the Java Virtual Machine – a portable technology for programs written in Java. Such a program is compiled only once to the so called bytecode. There is a dedicated Virtual Machine on each different platform which forms a kind of software interface between the Java bytecode and a native environment. Every time user works with the software the bytecode is run on this VM.
- Another possible solution is the implementation of Internet technologies like: HTML, ASP or JSP in a client-server form. Client side is constituted as a Web site that runs in a Web browser.

After some considerations Web browser occurred to be a good graphical environment on which the solution can be based.

- It is available to install on any computer with the Linux, Windows or Solaris operating system with graphical environment.
- Web browser as an end-user application enables distribution of software through the network, user can be notified about new versions.
- Web browser provides graphical context with well-known interface.

Solutions that utilize Internet technologies previously stated are:

1. Static Web pages, written in a pure HTML – presentation of an FSM is difficult, only in a form of matrix description or static pictures.
2. Dynamic Web pages in PHP, ASP or JSP technology – there is still a lack of elegant visualization.
3. Web pages extended by Java-Script – there exists possibility for animation and interaction but everything has to be translated and is processed as HTML.
4. Java Applet technology – enables almost the same complexity and functionality as ordinary Java applications what means that for example the same graphical libraries can be used as for standalone programs.

The advantages of applets decided that it is a best solution for a project. As any Java program, applets base on the Virtual Machine concept. The Java Virtual Machine is available for the previously specified platforms and is usually provided during standard installation process of a browser or easily available to download and install independently. Java development kit is an open environment what means no cost of purchase. Another advantage is that Java language becomes more and more popular. It is a new Object Oriented language which omits problems that harass older ones.

Ad. c) and d)

Graphical User Interface of an applet can be created using the same Java libraries which are used for applications, additionally graphical background is provided by the Web browser. The main advantages of using GUI were described in previous chapter. Well designed graphical interface provides intuitive interaction basing on perception of humans.

7.2 Client-server model

The Finite State Machine which was described in chapter 5 runs as a computer program on a server and provides some features for other computer programs interested in these functionalities.

To visualize an FSM a precise knowledge is needed about its structure and current state. These informations have to be received by the visualization program. Best solution to do this is to make it embedded in the FSM and provided on applet request. That defines the client – server model.

7.2.1 Server part

Information about the structure of the machine are hard to gain from the pure source code of FSM because all the dependencies introduced in chapter 5, that constitutes particular state automata, are hidden on the low level of abstraction, namely on the programming language level. Thus, additional work to establish this information was needed. It was done in the development environment of FSM – the Stateflow Toolbox. The information format was set to the popular standard of XML.

XML is an acronym for extensible markup language, is the standard of the World Wide Web Consortium. It was designed to describe data i.e. structure, store and send information. XML uses elements which are not predefined, only the syntax for an element is defined what allows anyone to create its own custom element.

Important feature of XML, that is very helpful in this project, is the fact that XML elements have relationships to each other. They can be nested what creates parent-child relation. In case of parallel elements the sister relationship exists. The same is true for the FSM states, therefore XML is a natural choice for the project.

XML elements are purely text strings. They are defined by pairs of tags – start tag: ‘<tag_name>’ and end tag ‘</tag_name>’. Between them is the place for element value. The start tag may also contain some attributes.

The example tag may look as follows:

```
<tag_name attribute1="some_value" attribute2="1234">Element Value</tag_name>
```

Two distinct informations are defined and described using XML standard in this project. These are **FSM_STRUCTURE** and **CURRENT_STATE** data.

FSM_STRUCTURE:

- Consists of main elements <STATE> and <TRANSITION> which represents main FSM’s objects which are states and transitions respectively.
- <STATE> tag includes attributes describing name of the state, unique id, its type, and decomposition – variables that define states in the Stateflow Toolbox. State tags can be nested if needed to describe the hierarchical dependencies.
- <TRANSITION> tag includes following attributes: id, label, source state name, source state id, destination state name, destination state id. Transition tags are not nested – there cannot exist such a dependency between them.
- An important fact is that there is no such a functionality of a Stateflow Toolbox that provides generation of description of an FMS diagram in a form of XML. The general Matlab script language was used to create a text file in an XML form. Connection to the Stateflow diagrams was possible through Stateflow API – dedicated for non-GUI manipulation of diagrams. Implementation details are described in the next chapter.

CURRENT_STATE:

- Contains a list of active states of a machine.
- Important assumption was stated by the FSM designer: names of states are distinct therefore the list defines active states in a unique way.

Another important designer assumption is that FSM structure cannot change during the operation of the automata. That allows the FSM_STRUCTURE information to be static data which is checked only once per a connection to state automata.

CURRENT_STATE data is dynamic, the list is updated according to the current state of FSM and is checked permanently during session.

7.2.2 Client part

As previously described, technology that best suits the project requirements, is the Java applet technology. Therefore client program was designed as applet.

It is divided into two modules strictly related to each other, namely: graphical framework and visualization module.

GRAPHICAL FRAMEWORK:

Graphical framework provides graphical interface for the following functions:

- connecting to FSM
- reading FSM structure
- refreshing FSM state on demand
- saving layout
- moving up in the state hierarchy

- FSM visualization by the dedicated module

VISUALIZATION MODULE:

This module is in charge of presentation the Finite State Machine in a form of directed graph, described in chapter 6. It allows clear differentiation of presented objects using color, labels or other distinctions.

These objects are:

- objects that group parallel states, so called flows
- objects that represents superior states – superstates
- objects representing ordinary states – simple states

As far as transitions are concerned there no need to differentiate them.

That part of applet description concerned visual presentation. Another important feature is interaction with user. It is realized on two levels: edit and work. Proper design and implementation of these phases make the final project more effective.

EDIT MODE

This phase is strictly connected with work phase and is a base for it. In edition mode only the graphical presentation of objects can be changed, in particular: size and placement, path of transitions. User cannot change number of state objects, their interconnection, number of transitions. Edition phase provides adaptation of graphical presentation that best suits user needs. Adjustment capabilities are so flexible that enables solution of layout problem. It is the basic problem that user happen to encounter working with such complicated graphs.

Due to the change of size of graphical objects, user extends the area from which the transitions start or in which they end. The view is better distributed across the screen and is effectively acquired by the user brain. The dependencies are read and understood

faster. Program enables switching into working mode and immediate usage of created layout.

WORK MODE

In this mode visualization system presents change of state of the machine. The interaction process is realized using mouse pointer and by clicking on the objects. The main response of the system while clicking on the flow object or superstate object is that the view is changed to present internal structure of this state. In work mode user perceives visual signals representing actual state of a machine. State boxes change colors to present its activity or inactivity.

8. Problem Solution - Implementation

8.1 MATLAB Stateflow part

8.1.1 Data structure in XML

XML data description was used to describe a structure of FSM designed in Matlab Stateflow Toolbox. This description was stored in a text file. Such a solution allows portability. Precisely defined FSM structure can be implemented in a program or in this case stored in the FSM server as its property.

Description of created XML elements is presented below, together with an example structure:

```
<?xml version='1.0' encoding='utf-8'?>
<FSM>
<STATES id="0" decomp="PARALLEL_AND">
  <STATE id="12" name="ON" type="simple"
    decomp="EXCLUSIVE_OR"></STATE>
    .
    .
    .
  <STATE id="434" name="WORK" type="super"
    decomp="EXCLUSIVE_OR">
    <STATE id="534" name="STAGE1" type="simple"
      decomp="EXCLUSIVE_OR"></STATE>
      .
      .
      .
    <STATE id="111" name="STAGE2" type="simple"
      decomp="EXCLUSIVE_OR"></STATE>
  </STATE>
</STATES>
```



```
<TRANSITIONS>
  <TRANSITION id="222" label="EVENT1"
    sid="434" sname="WORK" did="12" dname="ON">
  </TRANSITION>
  .
  .
  <TRANSITION id="54" label="EVENT2"
    sid="534" sname="STAGE1" did="111" dname="STAGE2">
  </TRANSITION>
</TRANSITIONS>
</FSM>
</xml?>
```

Main tag is **<xml>** which defines version and encoding of the file.

<STATES> and **<TRANSITIONS>** are the simple grouping tags. **<STATES>** tag represents the point of origin of the state hierarchy. It has the 'id' and 'decomp' attributes which will be described later.

The most important tags are:

- **<STATE>**
- **<TRANSITION>**

<STATE>

This tag has the following attributes:

- **id** - an unique id number of the state, set in the Stateflow environment to differentiate states.
- **name** - name of the state given by the FMS designer.

- type - possible two values:
 - simple - simple state is basic and atomic unit of the FSM
 - super - super state contains a set of simple states
- decomp - two values of decomposition are possible:
 - EXCLUSIVE_OR – the content of such a state is in a exclusive_or relationship, only one state can be active at a time.
 - PARALLEL_AND - defines that content of a state consists of flows, independent blocks of states, not related to each other.

<STATE>, as seen in the example, can be nested in each other to define parent-child relationship.

<TRANSITION>

This tag has the following attributes:

- id - an unique id number of the transition, set in the Stateflow environment to differentiate them.
- label - text string given in a Stateflow, containing definition of transition's event, functions, etc.
- sid – id number of the state that transition begins
- sname – name of the state that transition begins
- did - id number of the state that transition ends
- dname - name of the state that transition ends

8.1.2 MATLAB functions

Creation of such a XML structure is performed by dedicated functions written in a Matlab Script language, using Stateflow API [10]. This API allows user to create or manage state diagrams. It also allows reading of all properties of all elements used to build specified diagram.

This feature was utilized in the following Matlab functions:

states

Signature: result1=states(chartname,filename)

Input: chartname – character string defining name of the chart in a state diagram

filename – character string defining name of the new created file containing XML data

Output: result1 – equals 1 if function finishes without error

Functionality:

- creates main layout of xml file; puts header <xml> and opening tags <FSM>, <STATES>, <TRANSITIONS>
- invokes 'findState', 'findTrans' functions that search through a state diagram to find states and transitions respectively
- puts closing tags to create well-formed XML file

Listing 1 (states.m):

```
function res1=states(chartname,filename)  
%init  
fid=fopen(filename,'wt');  
rt=sfroot;  
m=rt.find('-isa','Simulink.BlockDiagram');  
chArr=m.find('-isa','Stateflow.Chart');
```

```
stRoot=chart.find('-depth',1,'-and','-isa','Stateflow.State')
%open
fprintf(fid,'%s\''%s\''%s\''%s\''%s\n','<?xml version='1.0',' encoding='utf-8','?>');
fprintf(fid,'%s\n','<FSM>'),
fprintf(fid,'\t%s','<STATES>');
fprintf(fid,'%s\''%s\''',' id='0');
fprintf(fid,'%s\''%s\''',' decomp='PARALLEL_AND');
fprintf(fid,'%s\n','>');
fprintf(fid,'\t\t%s','<STATE>');
.
.
%STATES
stArr=stRoot.find('-depth',1,'-and','-isa','Stateflow.State');
stArr=setdiff(stArr,stRoot);
if length(stArr)>0 nothing=findState(stArr,fid,glC); end;
.
.
%TRANSITIONS
fprintf(fid,'\t%s\n','<TRANSITIONS>');
trArr=stRoot.find('-isa','Stateflow.Transition');
if length(trArr)>0 noth=findTrans(trArr,fid); end;
fprintf(fid,'\t%s\n','</TRANSITIONS>');
.
.
%close
fprintf(fid,'%s','</FSM>');
fclose(fid);
resl=1;
```

findState

Signature: `glC = findState(stArr, fid, glC)`

Input: `stArr` – array of states

`fid` – file identifier

`glC` – global counter set in the ‘states’ function

Output: `glC` – modified global counter

Functionality:

- places <STATE> tags with proper attributes extracted from the state diagram and preserve defined relations between states.

Listing 2 (findState.m):

```
function glC = findState(stArr, fid, glC)
    stCounter = length(stArr);
    for i=1:stCounter,
        glC=glC+1;
        nth=tabx(glC, fid);
        fprintf(fid, '%s', '<STATE'),
        fprintf(fid, '%s\''%d\''', ' id=', stArr(i).Id);
        fprintf(fid, '%s\''%s\''', ' name=', stArr(i).Name),
        fprintf(fid, '%s\''%s\''', ' decomp=', stArr(i).Decomposition),
        stArr2=stArr(i).find('-depth', 1, '-and', '-isa', 'Stateflow.State');
        .
        .
        if length(stArr2)>0
            fprintf(fid, '%s\''%s\''', ' type=', 'super');
            fprintf(fid, '%s\n', '>');
            glC=findState(stArr2, fid, glC);
        end;
```

```
    nth=tabx(glC, fid);  
    fprintf(fid, '%s\n', '</STATE>');  
    glC=glC-1;  
end;
```

findTrans

Signature: noth=findTrans(trArr, fid)

Input: trArr – array of transitions

fid – file identifier

Output: noth – equals 1 if function finishes without an error

Functionality:

- places <TRANSITION> tags parallel to each other with proper attributes.

Listing 3 (findTrans.m):

```
function noth=findTrans(trArr, fid)  
    trCounter=trArr.length;  
    for k=1:trCounter,  
        fprintf(fid, '\t\t%s', '<TRANSITION>'),  
        fprintf(fid, '%s\ "%d"', ' id=', trArr(k).Id);  
        string=trArr(k).LabelString;  
        string2=rep(string, '&', '&');  
        string3=rep(string2, '<', '<');  
        string4=rep(string3, '>', '>');  
        fprintf(fid, '%s\ "%s"', ' label=', string4),  
        ...  
        fprintf(fid, '%s\ "%s"', ' sName=', trArr(k).Source.Name);  
        ...  
        fprintf(fid, '%s\ "%d"', ' dId=', trArr(k).Destination.Id);  
        ...  
    end
```

```
        fprintf(fid, '%s', '>'),  
fprintf(fid, '%s\n', '</TRANSITION>'),  
nth=1;  
end;
```

Additional function:

tabx

Signature: nth=tabx(glC, fid)

Input: glC – global counter

fid – file identifier

Output: nth – equals 1 if function finishes without an error

Functionality:

- Manage the number of 'tab' white characters that are used to format the code. Each level of nesting of states is visualized using different number of 'tabs' in front of <STATE> tags, creating easy to read tree-like structure.

Listing 4 (tabx.m):

```
function nth=tabx(glC, fid)  
    for j=1:glC,  
        fprintf(fid, '%s\t', ''),  
        nth=1;  
    end;
```

The possibility to obtain list of active states of FSM during its work, in XML form, was developed by FSM designer implementing proper functions: **ml.put(state_name)**, **ml.pop(state_name)**. Calls to this functions has to appear in the proper moment during

operation of FSM. It is the 'enter' procedure for 'put' function call and 'exit' procedure for 'pop' function call.

Additional script '**funext**' is in charge of placing these function calls into respective procedures. It traverses FSM structure and extends each of the state labels with calls: `ml.put(state_name)` and `ml.pop(state_name)`. This function prints out on the screen some status messages about the changed state.

funext

Signature: `a=funext(chartname)`

Input: `chartname` - character string defining name of the chart in a state diagram

Output: `a` - equals 1 if function finishes without an error

Listing 5 (funext.m):

```
function a=funext(chartname)
    rt=sfroot
    m=rt.find('-isa','Simulink.BlockDiagram')
    chArr=m.find('-isa','Stateflow.Chart')
    ...
    stArr=chart.find('-isa','Stateflow.State')
    for j=1:length(stArr)
        j
        val='changing following state:'
            stArr(j).Name
            stArr(j).Path
            stArr(j).LabelString
            stArr(j).LabelString=placer(stArr(j).LabelString,stArr(j).Name)
    end;
    a=1;
```


placer

Signature: result=placer(string,name)

Input: string – character string that is extended, the label of state

name – character string representing state name

Output: result – resulting character string.

Functionality:

- Performs a checking procedure that is implemented to put calls only once and in the proper procedures.

Listing 6 (placer.m):

```
function result=placer(string,name)
    ...
    result=string;
    en=strcat('ml.push_','',name,'');
    ex=strcat('ml.pop_','',name,'');
    ...
    index2=findstr(result,'entry:')
    if length(index2)==1 index2=index2(1); else index2=0; end;
    pushplace=findstr(result,en);
    if length(pushplace)==1 pushplace=pushplace(1); else pushplace=0; end;
    popplace=findstr(result,ex);
    if length(popplace)==1 popplace=popplace(1); else popplace=0; end;
    ...
    popplace=findstr(result,ex);
    ...
    result=[result '/' s2 'entry:' en s2 'exit:' ex];
    ...
    index1=findstr(result,'/');
    if result(index1(1)+1)~=s2 result=regexprep(result,'/','\n'); end;
```

```
index3=findstr(result,'during:');  
if length(index3)==1  
    if result(index3(1)-1)~=s2 result=regexprep(result,'during:','\nduring:'); end;  
end;  
index4=findstr(result,'exit:');  
if result(index4(1)-1)~=s2 result=regexprep(result,'exit:','\nexit:'); end;
```

8.2 Applet part

The whole implementation of the program will be easier to understand when its description is realized as a flowchart. All the implemented methods have the main goal to satisfy requirements of the project and fulfill the design rules defined in the Solution Structure chapter.

8.2.1 Applet framework

The top framework for this visualization program is the applet environment. It consist of an Internet Browser with applet-running function enabled. Such a browser is able to show HTML files and applets. Simple HTML file containing applet tag was created and stored on a dedicated server in the DESY network.

HTML file listing:

```
<html>  
  <head>  
    <title>Applet page</title>  
  </head>  
  <body>  
    <applet  
      code="Applet.class" archive="vizz.jar" width=600 height=400>  
    </applet>  
  </body>  
</html>
```

8.2.2 Graphical User Interface

Another level is constituted of Graphical User Interface, that provides the management part and main visualization area. GUI design follows the rules stated in [17].

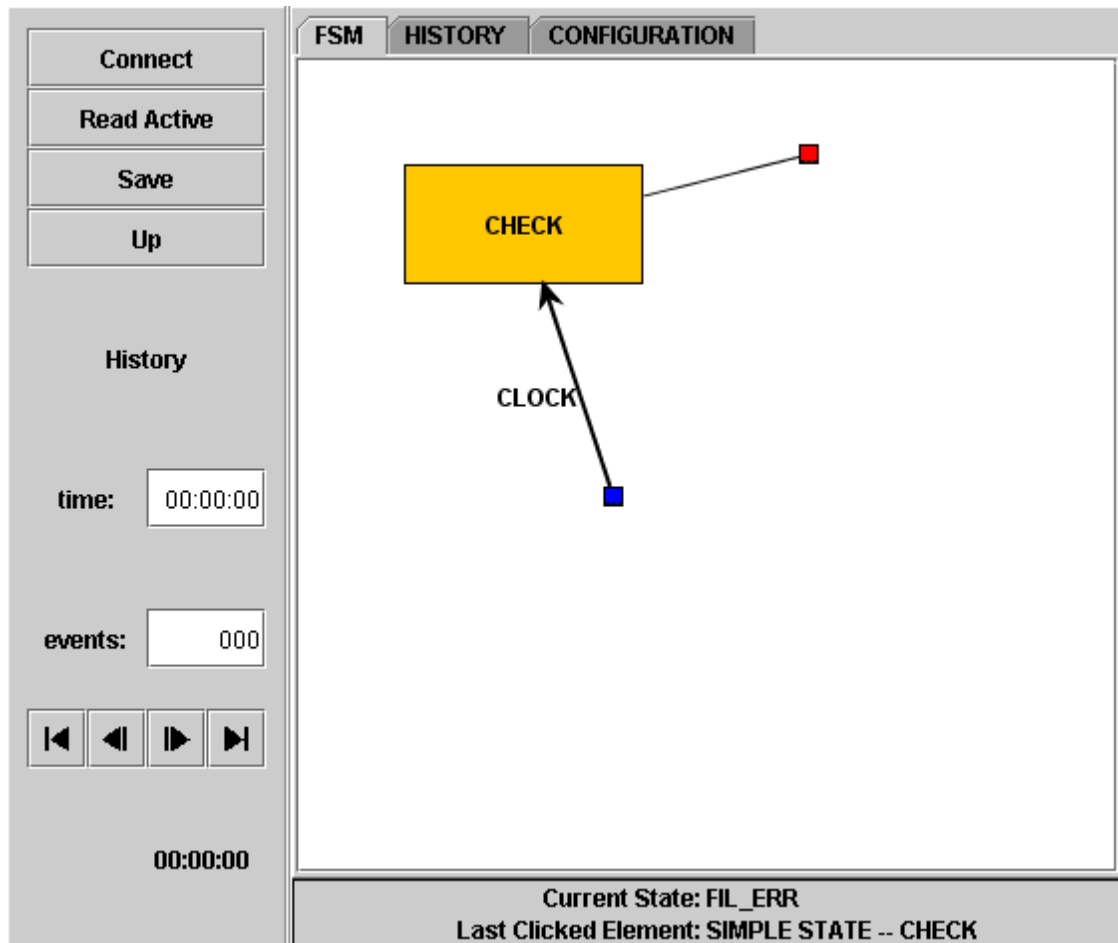


Fig. 8.1 Layout of GUI elements

GUI was implemented using the Java graphical libraries: javax.swing and java.awt. The GridBagLayout manager was chosen because it is sophisticated but flexible layout manager that aligns components by placing them within a grid of cells what allows designer to follow easily the rules stated in the Visualization chapter.

Management functions of the program during its main operation are provided by the set of buttons which implementation is listed below:

- Connect to FSM – performs the connection test
- Read Active States – immediate request of the list of active states
- Save layout – saves layout of the graph to a file
- Move Up – moving up in the states hierarchy

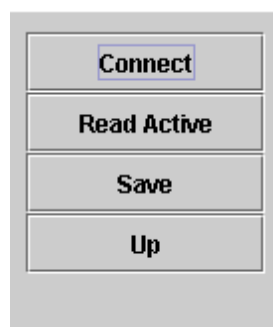


Fig. 8.2 Management buttons

Result of their work is presented in the status bar. That allows the end-user being notified of all the program behavior, errors, and suggestions.

Fig. 8.3 Status bar implementation

Additional part of management panel consists of elements used while working with the history panel. These are:

Time display – shows time of the last readout

Events display – shows number of read events

Button bar – used to step over events

Events labels – shows number and time of current event

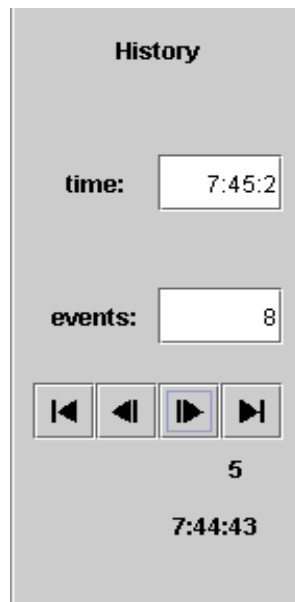
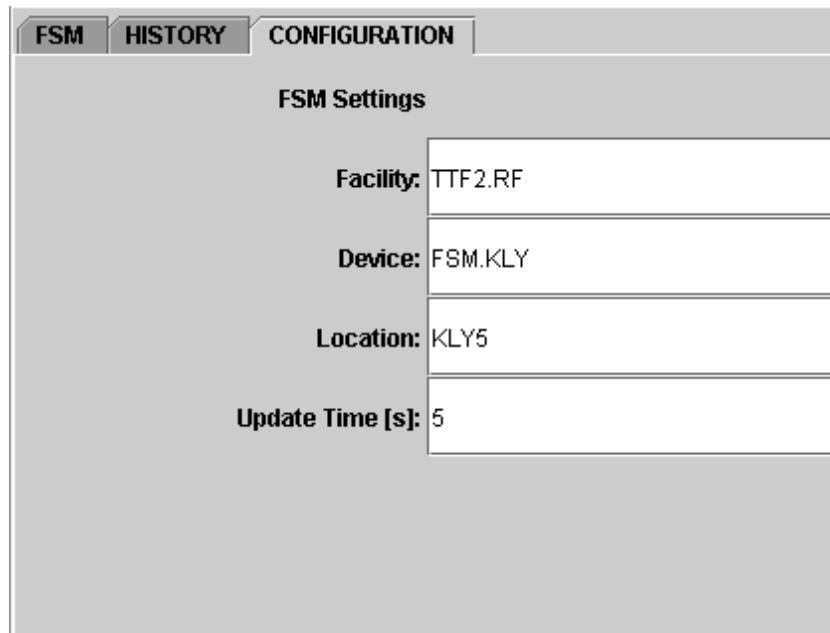


Fig. 8.4 History control buttons

The very important panel providing initial, basic configuration of the program is the 'Configuration Panel' presenting:

- a) FSM server address in the ENS form
- b) Time interval of readouts (in seconds)



The image shows a software interface with three tabs: 'FSM', 'HISTORY', and 'CONFIGURATION'. The 'CONFIGURATION' tab is active. Below the tabs, the title 'FSM Settings' is displayed. There are four input fields with labels: 'Facility:' containing 'TTF2.RF', 'Device:' containing 'FSM.KLY', 'Location:' containing 'KLY5', and 'Update Time [s]:' containing '5'.

Fig. 8.5 History control buttons

8.2.3 Visualization Area

Program's visualization engine works in this area. It presents, in the graphical form, an FSM that the program is connected to. The same FSM structure is shown in to different tabs: 'FSM' and 'HISTORY'. The external JGraph library was used to allow intuitive mouse operations on the graph representation and smooth and quick rendering of a graphics. It allows fast customization of a Graph Model using a wide range of predefined attributes. Additional JAddons library was utilized to provide the layering of a graph.

This graphical part, served by the **StateManager** class, works in a strictly defined way:

1. Parsing the XML data using the `javax.xml.parsers`, `org.xml.sax` and `org.w3c.dom` libraries.
2. creation of DOM object which gives: easy access to XML elements

3. reading the <STATE> elements to tempList1 of type: ArrayList.
4. reading the <TRANSITION> elements to transNodeList of type: ArrayList
5. Creation of ArrayList that contains objects of the type MyTrans which is the representation of the transition in the program - that ArrayList contains all the transitions that were defined in XML.
6. Calling the '**manage()**' method. It contains main algorithm of the program (points no. 7,8,9)
7. Creation of a JGraph and setting the parameters:
 - setAntiAliased – true
 - setClonable – false
 - setEditable – false
 - setDisconnectable – false
 - setSizable – true
 - setSelectionEnabled – true
 - setGridEnabled – false
 - setGridVisile – false
8. Check if the graph layout was previously saved
 - if true: graph is added to view panel and redrawn,
 - if false: the structure of state is checked (described in the point no.7)
9. INTERACTIVE PART in which mouse is served by the implemented MouseListener interface.
10. '**getInnerStructure**' method invocation to find out a content of a state (described in pts. no. 11-17). Its main task is to create the array of objects of a type DefaultGraphCell for the current state. These are the graphical elements representing both states and transitions with its parameters.

- parameters for states are: id, name, decomp, type
- parameters for transitions are: id, label, sid, sname, did, dname

11. ArrayList 'stateList' containing MyState objects is initialized. It lists all child states for a state that is currently being browsed. MyState is an object that represents states in the program.

12. ArrayList called 'tempTrans' is initialized. It contains state's transitions for the current node.

13. From this moment stateList and tempTrans are used.

14. Cleaning the tempTrans list:

- connecting states in a way that omits junctions
- removing the redundant transitions
- removing the doubled transitions
- adding 'default' state
- adding representation of external states

15. Creation of graph elements from the list of states and transitions

16. Recognition of proper state type and creation of proper graphical element, that meets the graphical code using the following attributes of DefaultGraphCell:

- setBorderColor
- setBackground
- setGradientColor
- setOpaque
- setBorder

17. Creation of transition elements between states. All the transitions are identical and their attributes are set as follows:

- setLineEnd – ARROW_CLASSIC
- setEndFill – true
- setBendable – true

- `setLineWidth` – 2
- `setExtraLabels` – true
- `setLabelAlongEdge` – false

18. Mouse interaction with the visualization module (described in the User Manual).

9 User Manual

9.1 Installation Instructions

Installation is done by copying the file *vizz.jar* to the user account with the access to the Distributed Object Oriented Control System – where the ENS address will be recognized. The next step is creation of the HTML file containing the applet framework presented in section 8.2.1.

The requirements concerning the installation platform are defined as follows:

1. The Java Runtime Environment installed (in version 1.4.2 or higher).
2. Web browser available (with the possibility to run applets).

9.2 User Operations

9.2.1 Running the program

1. The visualization program may be run as an applet in the web browser by opening the html file created in the installation phase.
2. Alternatively, it is possible to run the program as a standalone application by issuing the command: `java -cp . -jar vizz.jar`

9.2.2 User Interaction

In the major number of cases user operates with this program by means of mouse interaction. Only the configuration phase is done by means of a keyboard. The following steps provide a brief description of the order of actions user has to undertake to accomplish a given task.

The main usage scenario is as follows:

1. After running the program the visualization area is empty:

Fig. 9.1 Empty visualization area

2. Click the configuration tab and provide the ENS address divided into 3 fields: facility, device, location – that defines the FSM server in the DOOCS system. Specify the update time of readouts of active states. The default values are provided as shown in fig. 9.2.

Fig. 9.2 Configuration tab

3. Click on the 'Connect' button – the program connects with the address specified and reads the structure of the investigated FSM which can be seen in the 'FSM' and 'HISTORY' tabs. The hierarchy of a structure always starts with the 'ROOT' flow visible.

Fig. 9.3 Top view of a structure

4. Entering the ROOT by double clicking the rectangular object one explores the whole fsm structure consisting of the following graphical elements:

Flow element (superstate) – the state with a PARALLEL_AND decomposition

Super State element – contains some simple states with EXCLUSIVE_OR decomposition

Simple State – basic finite state machine element (does not contain any internally nested structure)



Active State

Transition – with corresponding label

Default state representation – red box pointing to a default state

Loop transition representation

5. Change the size or placement of graphical objects to best utilize the visualization area. This is done by means of handlers which can be dragged with the left mouse button pressed. Right clicking on the transition element (arrow) adds additional handler that enables more sophisticated configurations This is shown on a picture below:

Fig. 9.5 Manually reconfigured fsm layout

6. Click the save button that saves the current layout of a state. It is used from now on to define a layout for a particular state.
7. Click the 'Read Active' button – it starts the periodical updates of the active states with the time interval specified on the configuration tab. Any further use of 'Read Active' button force the immediate check of the active states. Program is now ready and the user can browse the FSM structure and follow the path of active states.
8. The visible change after invoking the 'read active' procedure is also the change in 'events display'. The number of events increases when the new active states are different from previously read. The 'time display' updates to present the time of the last readout.

9. Additional functionality is provided by the 'HISTORY' tab in which one can view changes of active states. It is supported by means of events control buttons. This action is reflected in the change of 'event labels' showing the time and number of the investigated event.

Fig. 9.6 Event control buttons

10 Conclusions and Summary

The main goal of this master thesis was to develop a computer program to monitor work of Finite State Machine and showing its internal structure.

Requirements for this project were mainly based on the experience and needs of the FSM designer and operators. Mainly during tests in real or test environment, there is a need for tool that would:

- show the internal structure of an FSM
- be able to provide an information about its current state, the state that it was changed from and parameters that triggered this transition;
- be independent from any configuration files or source code of an FSM.
- record history of changes

This features would greatly improve work of a designer and decrease time spent on testing.

The issue of efficiency and interface being intuitive is also important for the operator of the control system. The one who will be responsible for the operation of the system, could faster react in case of problems and minimize the break-time because visualization gives better understanding of the current status of FSM. These requirements can be satisfied by implementation of a Graphical User Interface with the strong stress put on the visualization of an FSM.

Reaching the best solution of a stated problem required studies of different programming concepts and technologies.

The issue that the program has to be available from any computer in the local network required knowledge about multiplatform solutions. Here the Java Virtual Machine was chosen which is a portable technology for programs written in Java. Such a program is

written and compiled only once then interpreted by the platform specific Virtual Machine saving time spent on considering platform dependent issues.

The further extension to the concept of a multiplatform program was the utilization of Internet technologies – namely Java Applets and Web browsers. Browsers enable distribution of software through the network, user can be notified about new versions of program. They provide graphical context with well-known interface.

The concept of Graphical User Interface follows the rules that governs human behavior and cognition. Understanding of that rules helps in proper design of intuitive interface.

The theory of Finite State Machines was necessary to define the object of visualization. Particular implementation of FSM in a form of statechart was considered. Such a implementation is used in the Stateflow Toolbox used by the FSM designer. To export the structure of FSM to the server a flexible but simple format was created based on the standard XML tags. The usage of Stateflow API and Matlab scripts language enabled fast creation of tools that make this export possible.

The main part of a program – visualization system had the task of creation of intuitive representation of an FSM from the XML structures generated in the Stateflow. No previous knowledge about the internal structure of a machine could be obtained because system does not depend on any configuration files. The FSM topology was defined in the XML in a precise way that enabled proper reconstruction of the automata. Another problem was the graphical library used to render the graphical elements of an FMS representation. First attempt was to write own library, utilizing the concept of a directed graph but another problem was the layout of the graph. Simple solution was the circular layout with all states placed on the circle circumference but in case of a graph with a lot of transitions (almost each to each another) it became unreadable. Therefore the choice was made to use some open source library, but already developed and in a stable version – namely JGraph. The problem of layout was solved by giving the user the advanced tool that allows customization of FSM view and saving it. Because the graphs are rarely planar user invention is better than a computer algorithm.

The strong advantage of a program is the history panel in which the whole FSM work is stored and can be viewed step by step.

The hard task during the development of the program was to perform tests in the real environment because of the reduced interaction quality using the remote X-server connection from Lodz to Hamburg.

The proposal for the next version of the program would be the extension of configuration options that would give better sense of control over the visualization system. Also the possibility to save the composed graph layout to a file is desired. This option would be helpful in case of multiple sessions with the same FSM.

11 References

- [1] SIXTH FRAMEWORK PROGRAMME, Structuring the European Research Area Specific Programme, RESEARCH INFRASTRUCTURES ACTION, *Annex 1 - "Description of work"*, RII3-CT-2003-506395
- [2] ILC- International Linear Collider, DESY, <http://www.desy.de/pr-info/desyhome/html/presse/hginfos/tesla/alles.en.html#project>
September 2005
- [3] S. Goloborodko, G. Grygiel, O. Hensler, V. Kocharyan, K. Rehlich, P. Shevtsov, DOOCS: an Object Oriented Control System as the Integrating Part for the TTF Linac, DESY Hamburg,
- [4] K. Rehlich, The TTF VUV-FEL Control System: DOOCS, DESY, 2005
- [5] Zadania DMCS, <http://desy.dmcs.p.lodz.pl/zadania/index.php>, August 2005
- [6] Z. Kohavi, Switching and the Automata Theory, McGraw-Hill, 1978
- [7] A. Gill, Introduction to the Theory of Finite-state Machines, McGraw-Hill, 1962
- [8] Y. Hardy, W. Steeb, Classical and quantum computing, Birkhauser Verlag, 2001
- [9] A. Tornambe, Discrete-event system theory: an introduction, World Scientific Publishing, 2005
- [10] Stateflow, MathWorks,
<http://www.mathworks.com/access/helpdesk/help/toolbox/stateflow/>, July 2005
- [11] D. Harel, Statecharts: A visual Formalism for complex systems, The Science of Computer Programming, 1987
- [12] FSM, Wikipedia, http://en.wikipedia.org/wiki/Finite_state_machine,
September 2005
- [13] Alexander Brandt, Improvements in LLRF control. Algorithms and Automation, DESY, 2004
- [14] B. Kosęda, W. Cichalewski, FSM Requirements for klystron, proposed solutions, use cases, boundary of the system., DESY, 2004

- [15] W. O. Galitz, *The Essential guide to User Interface Design*, John Wiley & Sons, 2002
- [16] C. Ware, *Information Visualization: Perception for Design*, Morgan Kaufman Publishers, 2000
- [17] I. E. Gordon, *Theories of Visual Perception*, Psychology Press, 2004
- [18] L. E. Wood, *User Interface Design: Bridging the Gap from User Requirements to Design*, CRC Press LLC, 1997

12 Appendix

CD-ROM content:

The attached CD contains two folders:

1. *matlab-scripts* –contains Matlab scripts used to generate the XML files from the Stateflow model and to extend the model with proper functions, it contains also the *howto.txt* file with the manual for users of these functions.
2. *visualization* – contains main files for visualization program, subfolder *src* contains java source files, subfolder *docs* contains manual files about starting the application and signing applets.