TALLINN UNIVERSITY OF TECHNOLOGY

Department of Computer Engineering

IAY84LT

WEB-BASED SYSTEM FOR FINITE STATE MACHINES DECOMPOSITION

BY

SERGEI DEVADZE

A Master's Thesis
Submitted to the chair of Digital Systems Design
of the department of Computer Engineering

In fulfilment of the requirements for the
Degree of Master of Science
in Computer Engineering

Tallinn 2004

AUTOMAATIDE DEKOMPOSITSIOONI VEEBIKESKKOND

Koostaja: Sergei Devadze

Annotatsioon

Lõplike automaatide dekompositsioon on oluline probleem digitaalskeemide projekterimise valdkonnas. Mitmed dekompositsioonimeetodid ja lahendused olid välja pakutud digitaalsüsteemide sünteesi optimeerimiseks. Üks nendest, mis on esitatud selles töös, oli välja töötatud Tallinna Tehnikaülikooli digitaaltehnika õppetoolis.

Käesoleva töö käigus on loodud veebikeskkond, mida saab kasutada lõplike automaatide dekomponeerimiseks. Väljatöötatud tarkvara on valmis rakendamiseks nii dekompositsiooni teaduslikul uurimisel kui ka interaktiivse õppevahendina. Süsteem pakub ka raamistiku, mida võiks kasutada tarkvara parendamiseks ja uue tarkvara loomiseks. Süsteemi kasutajad võivad saada juurdepääsu dekompositsioonikeskkonnale Interneti kaudu.

Juhendaja: Dots. Aleksander Sudnitsõn

Abstract

Decomposition of Finite State Machines (FSMs) is an important task in the field of design of digital systems. The problem of decomposition was studied quite thoroughly and various methods were proposed in order to improve its efficiency. One of the methods, that was elaborated at Tallinn University of Technology, can be used for optimization of implementations of high-complexity FSMs as well as for low-power synthesis.

This thesis presents a newly developed web-based environment that implements investigated approach of decomposition of FSMs. The developed software can be used as a research instrument in the area of decomposition or as an educational module. In addition, the system contains the powerful framework that can be used for developing new software in this area or improving functionality of already created environment. The system has interactive nature and can be easily accessed over Internet.

Table of Contents

LIST OF PUBLICATIONS ... 6
LIST OF ABBREVIATIONS ... 7
1 INTRODUCTION ... 8
1.1 Thesis Contribution ... 8
1.2 Thesis Contents ... 9
2 THEORETICAL BACKGROUND ... 10
2.1 Finite State Machines ... 10
2.2 Machines Network ... 12
2.3 Partitions and Partition Pairs ... 13
2.4 Partitions with Don’t Care’s ... 14
2.5 The concept of Informational Relationship Measure ... 15
2.6 Decomposition of FSM ... 15
2.6.1 Choosing a system of partitions on the set of states of FSM ... 17
2.6.2 Coding of the network ... 18
2.6.3 Determining of the structure of the network ... 19
2.6.4 Defining of the basis of the network ... 19
2.7 Partitions Search Problem ... 20
3 DECOMPOSITION SOFTWARE ... 22
3.1 Overview ... 22
3.2 Graphical Interface Tool ... 23
3.2.1 Interface description ... 24
3.2.2 Loading source FSM ... 26
3.2.3 Specifying set of partitions ... 26
3.2.4 Decomposition ... 28
3.2.5 Probabilities computing ... 29
3.2.6 Built-in libraries ... 30
3.3 Command-line Tools ... 30
3.3.1 Decomposition ... 31
3.3.2 Partition search ... 33
3.3.3 Generation of random set of partitions ... 34
3.3.4 Checking format of the FSM ... 36
3.4 Software API ... 36
3.4.1 API Overview ... 36
3.4.2 The API Core Classes ... 37

4 CONCLUSIONS 39
4.1 Thesis Summary 39
4.2 Future Work 40
5 REFERENCES 41
APPENDIX A CONFIGURATION FILES FORMAT 42
APPENDIX B DATA FORMATS 45
APPENDIX C SOFTWARE BUNDLE ON INCLUDED CD 52

List of Publications

1. S. Devadze, E. Fomina, M. Kruus, A. Sudnitson. *Web-Based System for Sequential Machines Decomposition* // Proc. of IEEE EUROCON 2003 International Conference on Computer as a Tool, Ljubljana, Slovenia, 2003, vol. 1, pp. 57-61
2. S.Devadze, R.Gorjachev, A.Jutman, E.Orasson, V.Rosin, R.Ubar. *E-Learning Tools for Digital Test* // Proc. of III International Conference 'Distance learning – educational sphere of XXI century', Minsk, Republic of Belarus, 2003, pp. 336-342
3. S.Devadze. *Web-Based Training System for Teaching Digital Design and Test* // Proc. of 7th International Student Conference on Electrical Engineering (POSTER'2003), Prague, Czech Republic, May 2003
4. S. Devadze, A. Jutman, A. Sudnitson, R. Ubar, H.-D. Wuttke. *Teaching Digital RT-Level Self-Test Using a Java Applet* // Proc. of 20th IEEE NORCHIP Conference 2002, Copenhagen, Denmark, 2002, pp. 322-328.
5. S. Devadze, A. Jutman, A. Sudnitson, R. Ubar, H.-D. Wuttke. *Java Technology Based Training System for Teaching Digital Design and Test* // Proc. 8th International Baltic Electronics Conference (BEC'2002), Tallinn, Estonia, 2002, pp. 283-286
6. S. Devadze, A. Jutman, M. Kruus, A. Sudnitson, R. Ubar. *Web Based Tools for Synthesis and Testing of Digital Devices* // Proc. International Conference on Computer Systems and Technologies (CompSys'2002), Sofia, Bulgaria, 2002, vol.1, pp. 91-96
7. S. Devadze, A. Jutman, A. Sudnitson, R. Ubar. *Web-Based Training System for Teaching Basics of RT-level Digital Design, Test, and Design for Test* // Proc. 9th International Conference Mixed Design of Integrated Circuits and Systems (MIXDES'2002), Wroclav, Poland, 2002, pp. 699-704
8. S. Devadze, M. Kruus, A. Sudnitson. *Web-Based Software Implementation of Finite State Machine Decomposition for Design and Education* // Proc. of International Conference on Computer Systems and Technologies (CompSys'2001), Sofia, Bulgaria, 2001, vol. 4, pp. 1-7

List of Abbreviations

API	Application Programming Interface
BLIF	Berkeley Logic Interchange Format
CAD	Computer-Aided Design
FSM	Finite State Machine
GUI	Graphical User Interface
JRE	Java Runtime Environment
JVM	Java Virtual Machine
PDC	Partition with Don’t Care’s
STG	State Transition Graph
STT	State Transition Table
URL	Uniform Resource Locator
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit
VLSI	Very Large Scale Integrated circuits

1 Introduction

It is well known that Finite State Machines (FSMs) are important components of design of digital systems and therefore methods that allow their efficient implementations are always of great interest. Decomposition is one of the design techniques that can significantly increase the quality of FSM synthesis.

Over the years the task of decomposition of FSM remains a classic problem of discrete system theory. In general, decomposition is the process of dividing of one machine into a network of two or more of interconnected and interacting sub-machines in such manner, that the terminal behaviour of network and source machine will be the same. The task of FSM decomposition has several goals. One goal is to make the process of synthesis easier by splitting the synthesis problem into several sub-problems that can be solved more efficiently. The other goal is to make synthesized structure meet the specific design constraints. There are also attempts to use decomposition for realization of low-power circuits.

The original decomposition approach described in this thesis was elaborated at Department of Computer Engineering of Tallinn University of Technology and is intended for optimization of synthesis process of high complexity FSMs as well as for design of low power applications [10], [11].

1.1 Thesis Contribution

The main goal of the presented work was to create a specific software that will realize investigated methods of decomposition. The software developed while working on this thesis provides the powerful framework for implementations of various tasks of FSM synthesis. Moreover, the interactive environment that was built on the basis of this framework can be used as a research tool in the field of decomposition as well as for education purposes. This means that on one hand the developed software is powerful enough for carrying out scientific experiments, but on the other hand has user-friendly

interface and interactive nature that allow to use it as an education module. In addition, the system has built-in support of various data formats in order to provide a connection with other tools in this area (such as CAD tools). The another major advantage of the developed environment is possibility to use it directly over Internet.

1.2 Thesis Contents

The remainder of the presented thesis is organized as follows. Chapter 2 describes basic theoretical aspects of the theory of sequential machines and methods of decomposition of FSM. Chapter 3 presents the detailed description of practical part of work and introduces the developed software. Chapter 4 concludes the thesis and outlines possible directions for future work.

2 Theoretical Background

In this chapter the main theoretical concepts and notations that were used as a basis of this work are presented. The chapter describes general issues of theory of sequential machines (which provide mathematical models for discrete computing devices with finite memory) and methods of decomposition of these devices.

2.1 Finite State Machines

Over the years, many important problems in sequential circuit systems and optimization have been approached using concepts from automata theory. The concept of *Finite State Machine* (FSM) is a widely recognized model that is used for representing the work of sequential digital devices. Because of their finite nature, FSMs yield better to describe, to analyze and to synthesize of intricate digital systems than any other alternative model.

Traditionally, FSM is represented by algebraic model consisting of a set of states, an input and output alphabets, a transition function that maps inputs symbols and current states to a next state and an output function that produces an outputs. Based on the level of abstraction the two different representations of FSM can be defined. An *abstract* automaton deals with "idealized" time and abstract nature of input and output signals that can be considered as symbols of some alphabets. A *structured* automaton, on the contrary, has finite number of inputs and outputs. To each of the inputs of structured FSM the symbol of an structure input alphabet can be assigned and the symbol of structure output alphabet can appear at each of its outputs. Usually, the binary alphabet (that consists only of two symbols: 0 and 1) is used for representing input and output signals of FSM.

The generalized definition of the FSM presented in [1] combines two previous statements of automata:

Definition 1 A *Mealy-type Finite State Machine* (FSM) is a quintuple (S, I, O, $\mathbf{d}$, l):

- $S=(s_1, s_2, \ldots, s_M)$ is a finite nonempty set of *internal states* (often called simply states) of the FSM
- $I=(x_1, x_2, \ldots, x_L)$ is a finite nonempty set of *input states* of the FSM, which either are symbolic or are represented as a binary vector of values of its *input signals*
- $O=(y_1, y_2, \ldots, y_T)$ is a finite nonempty set of *output states* of the FSM, which either are symbolic or are represented as a binary vector of values of its *output signals*
- $\mathbf{d}(x, s)$ is a relation from the (input state, present state) pairs to the next states (i.e., $\mathbf{d} \subseteq X \times S \times S$)
- $l(x, s)$ is a relation from the (input state, present state) pairs to the output states (i.e., $l \subseteq X \times S \times \mathrm{O}$)

Functions $\mathbf{d}$ and l are multiple-output Boolean functions.

An FSM is *deterministic* if both $\mathbf{d}$ and l are *functions*. A deterministic FSM is *completely specified* if both $\mathbf{d}$ and l are defined for all elements of their domains. An FSM is a *Moore-type FSM* if its output function l is depended only on the its internal state.

The next state function and output function of an encoded FSM are Boolean functional vectors. For a given circuit with L inputs, T outputs, and N flip-flops, the next state function is $\mathbf{d}$: $D(\mathbf{d}) \rightarrow S$ is a multiple valued next state function with domain $D(\mathbf{d})=D^L \times S^M$ and codomain S^M; $D_i=(0, 1)$ represents a set of values (symbols) each input variable x_i may assume.

Thus, the domain of $\mathbf{d}$ can be divided into two Boolean subsets, D^L corresponding to the input space, and S^M corresponding to the state space. The variables spanning the input space are associated with the primary inputs of the circuit and are called *primary input variables*. The variables spanning the state space are associated with the output lines of the flip-flops and are called *present state variables*. The variables spanning the codomain are associated with input lines of the flip-flops and are called the *next state variables*. Similarly, the output function is:

$l: D(l) \rightarrow R(l)$ is an output function with domain $D(l)=D(d)$ and codomain $R(l)=E^T$, $E_i=(0, 1)$ represents a set of values each output variable y_i may assume.

The variables spanning the codomain are associated with the primary outputs of the circuit and are called *primary output variables*.

Input condition is a Boolean function, a_{pq}, which is equal to 1 when FSM makes the transition from the state s_p to state s_q. Output signal is microinstruction, b_{pq}, the list of output signals which are equal to 1 on the transition of the FSM from s_p to s_q. The search for the next state and corresponding output (microinstruction) means the evaluation of the Boolean functions a on the Boolean space $(0, 1)^L$.

An FSM can be represented by two equivalent structures: a *State Transition Graph* (STG) and a *State Transition Table* (STT). The first is graphical and second is tabular representation forms. The nodes of STG are corresponded to the states of FSM while the directed edges represent transitions. In the tabular representation form each transition is presented by a separate row of table, while columns contain present and next states of FSM, input conditions and output signals corresponded to each of transitions.

2.2 Machines Network

Below the definition of network of FSM's is presented.

Definition 2 A *FSM's network* we treat as a system $N=(X_N, S_N, Y_N, R, g)$, where:

§ $X_N=(x_1, x_2, \ldots, x_k)$ is a set of network input symbolic variables

§ $S_N=(A_i \mid i \in I=(1, \ldots, n))$ is a set of state machines referred as component machines

§ $R \subseteq S_N \times S_N$ is a relation of connection (network structure)

§ $Y_N=(y_1, y_2, \ldots, y_u)$ is a set of network output symbolic variables

§ $g: (\times S_j) \rightarrow (0,1)^u$ is an output function of the network.

To describe the network more thoroughly, we use the set of internal symbolic variables of net $Z=\{z_i \mid z_i \in S_i,\ i \in I=\{1,\ldots, n\}\}$ and representation of relation of connection R as

incidence matrix $//r_{ij}//$. $r_{ij}=1$ means i-th component sub-FSM receives information from j-th component sub-FSM.

2.3 Partitions and Partition Pairs

This sub-section we start with standard definition of the notion of a partition [1].

Definition 3 A *partition* p on S is a collection of disjoint nonempty subsets of S whose set union is S, i.e. $p=(B_a)$ such that $B_a \cap B_b = \varnothing$ for $a \neq b$ and $\cup(B_a)=S$.

We refer to the sets of p as blocks of p and designate the block which contains element s by B_p(s). The notation $s \equiv t(p)$ means that elements s and t are contained in the same block of p.

Partition that contains exactly one element from S is called *zero partition*. Partition is called *unit partition* if it consists only of one block. Since a partition on a finite set can be interpreted as an algebraic form of the notion of information, the zero partition contains maximum information while the unit partition contains minimum information about the set.

If p_1 and p_2 are partitions on S, then:

- Product partition (i.e. $p_1 \times p_2$) is the partition on S such that $s \equiv t(p_1 \times p_2)$ if and only if $s \equiv t(p_1)$ and $s \equiv t(p_2)$.
- Sum partition (i.e. p_1+p_2) is the partition on S such that $s \equiv t(p_1+p_2)$ if and only if there exists a sequence in S $s=s_0, s_1, \ldots, s_n=t$ for which either $s_i \equiv s_{i+1}(p_1)$ or $s_i \equiv s_{i+1}(p_2)$, $0 \leq i \leq n-1$.
- Partition p_1 is *less than or equal to* another partition p_2 if and only if for every block B_i of p_1 there exist such block B_j in p_2 that $B_i \subseteq B_j$. Thus if $p_1 \leq p_2$ then $p_1 \times p_2 = p_1$ and $p_1+p_2=p_2$.

The basic research object in structure theory of sequential machines is a partition pair. To study of machine structure is begun in following definition with a formal notion of concept “information dependence”.

Definition 4 A *partition pair* (p, p') on the machine $A=(S, I, O, d, l)$ is an ordered pair of partitions on S such that $s \equiv t(p)$ implies $d(s, x) \equiv d(t, x)(p')$ for all x in I.

Thus (p, p') is a partition pair (p.p.) on S if and only if the blocks of p are mapped into the blocks of p' by S. That is, for every x in I and B_p in p, there exists a $B_{p'}$ in p' such that $\delta(B_p, x) \subseteq B_{p'}$. The concept of partition pairs based on the idea, that the first partition in a pair has enough information to calculate the second one.

Definition 5 If p is a partition on S of A, let $m(p)=\prod(p_i|(p, p_i)$ is a p.p. on A) and $M(p)=\Sigma(p_i|(p_i, p)$ is a p.p. on A).

Informally speaking, for a given partition p, the partition $m(p)$ describes the largest amount of information which we can compute about the next state of A knowing only p (i.e. the block of p which contains the present state of A). Similarly, for a given partition p', the partition $M(p')$ describes the least amount of information we must have about the present state of A to compute p' for the next state.

2.4 Partitions with Don't Care's

Definition 6 *A Partition with Don't Care's* (PDC) r of a set S is a collection of disjoint nonempty subsets of S. The disjoint subsets are called blocks of r and their set union is equal to S_d Í S. The set difference $S \setminus S_d$ is Don't Care's area of the PDC and can be considered as some distinguished (special) block *bc* which may be empty.

The PDC r in reality defines a set of conventional partitions, denoted by $G(\rho)$, generated by distributing the elements of distinguished block over the other blocks of the PDC and over new created blocks in all possible ways.

In case of PDC, the notation $s \sim t\ (\rho)$ means that both s and t are contained in the same non-special block of ρ. The relation *less than* or *equal to* between two PDC's ρ_1 and ρ_2 ($\rho_1 \leq \rho_2$) is exist if and only if for every non-special block B_i of ρ_1 there exists a non-special block B_j of ρ_2 such that $B_i \subseteq B_j$.

The operations on the Partitions with Don't Care's are defined in similar way as the operations on ordinary partitions. The set of all PDC's of S forms a distributive lattice [1] under this ordering with "+" (least upper bound) and "·" (greatest lower bound) operations such that ($\rho_1 \cdot \rho_2$) is the PDC on S such that $s \sim t\ (\rho_1 \cdot \rho_2)$ if and only if $s \sim t\ (\rho_1)$ and $s \sim t\ (\rho_2)$, $(\rho_1+\rho_2)$ is the PDC on S such that $s \sim t\ (\rho_1+r_2)$ if and only if there exists a sequence $s=s_0, s_1,\ldots,s_n = t$ for which either $s_i \sim s_{i+1}\ (\rho_1)$ *or* $s_i \sim s_{i+1}\ (\rho_2)$.

2.5 The concept of Informational Relationship Measure

This sub-section contains definitions of informational measure (formerly *inforesource*) of relationship between partitions.

Definition 7. *The information measure* $I(p)$ for partition $p(S)$ is $I(p) \overset{Def}{=} \log_2 \|p\|$.

This measure of separate partition $p=(B_i)$ gives exact meaning of minimum number of binary channels for parallel code transmission of information about the set S to partition $p(S)$. Thus, the partition's info resource is some information about the number of block of partition $p(S)$ or simple some information about partition $p(S)$.

From practical point of view, we are interesting in info resource of one partition relation to second partition.

Definition 8. *The information relationship measure* $I(p_2/p_1)$ of partition $p_2(S)$ relation to partition $p_1(S)$ is $I(p_2/p_1) \overset{Def}{=} \min\limits_{p_1 \cdot p_3 \leq p_2} I(p_3)$.

A partition $p_3(S)$ is such partition which reflects missing information of partition $p_1(S)$ to partition $p_2(S)$ (Figure 2.1). The minimum number of binary channels for transmission of this information is equal $\left] \min\limits_{p_1 \cdot p_3 \leq p_2} I(p_3) \right[$.

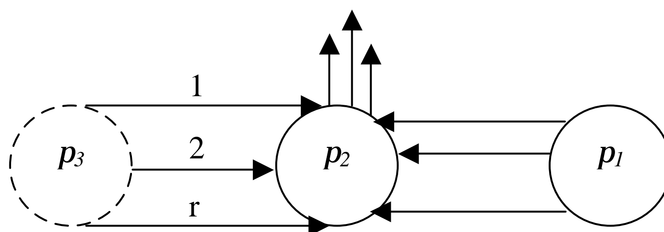

Figure 2.1 – Schematic illustration of concept of information relationship measure

2.6 Decomposition of FSM

It is not an overstatement to tell that the decomposition task is one of the most intricate and actual problems at complex discrete devices synthesis. Commonly, the FSM decomposition task is representing of prototype FSM as its network realization. It

means that, we construct such network of interconnecting and interacting component automata that it must realizes the work of source FSM.

Let p_i be the state partition inducting *state decomposition* on machine A. For each p_i, there is an associated FSM, A_i=<S_i, X_i, Y_i, d_i, l_i> where d_i and l_i is the partitioned next state and output functions consequently. The primary inputs are shared by the A_i's and the communication among A_i' is through state variables. The set (A_i | $i \in I$=(1, …, n)) of all the sub-FSM's A_i's represents the *decomposed network of FSM's* obtained from the original machine A when its set of states is decomposed according of p_i.

As mentioned above, on the set of states S of the source original FSM A we choose a set of state partitions p_i. Next we define some information partitions which are induced on A by a network that defines A. These "associated" partitions on A may be thought of as a global characterization (on A) of the information used and computed in a component machine of network. It is natural correspondence between local and global properties that allows us to approach the structure of machines with partition pair algebra.

The main condition of general FSM's decomposition is equality to zero of product of all selected partitions on the set of the states of the FSM. These partitions are called *complete set of partitions*. From informational point of view, while a partition on the set of states of source FSM is some measure of information about corresponding component sub-FSM, the zero partition on the set of states of decomposed FSM contains complete information about it.

The procedure of decomposition described in this section is based on the general form of decomposition without the restriction on their interconnection. Each sub-machine corresponds to a partition on the set of state. The procedure is illustrated on example of decomposition of the machine described by Table 2.1. Next we illustrate general description of main steps of the procedure.

Present state (s_p)	Input condition α_{pq}	Next state s_q	Output signal β_{pq}
1	x_1^x_2	1	y_1y_2
	^x_1^x_2	3	y_2
	^x_1x_2	5	y_5
	x_1x_2	6	y_1
2	x_2	2	y_2y_6
	^x_2	7	y_5
3	x_4	1	y_3
	^x_4^x_5	4	y_3y_4
	^x_4x_5	7	y_4
4	x_3	3	$y_2y_5y_6$
	^x_3	5	y_5
5	x_6	5	y_7
	^x_6	9	y_7
6	x_1x_3	7	y_5y_6
	^x_1x_3	8	y_5
	^x_3	9	y_6
7	x_2	8	y_2
	^ x_2	9	y_1y_6
8	x_5	3	y_3y_4
	^x_5	8	y_4
9	x_6	1	y_7
	^x_6	7	$y_3y_4y_7$

Table 2.1 – State Transition Table of example FSM

2.6.1 Choosing a system of partitions on the set of states of FSM

On this step of the decomposition procedure we select a complete set of partitions on the set of states. In our example the set of partitions will be the following:

$$p_1 = \{\overline{1,3,5,6}; \overline{2,7,8,9}; \overline{4}\}$$

$$p_2 = \{\overline{1,4,7}; \overline{2,6}; \overline{3,8}; \overline{5,9}\}$$

The one of the techniques of search of set of partition presented in Chapter 2.7.

Let us label the blocks of partition p_1 by a_1, a_2, a_3 and the blocks of partition p_2 by b_1, b_2, b_3, b_4 correspondingly.

2.6.2 Coding of the network

The coding of the network (global states of the net) gives us a set of internal binary variables of the network *Z*. Consider a set of states *S* and an encoding function $e: S \rightarrow \{0, 1\}^c$, for a given *c* (encoding length), that to each symbol $s \in S$ a code, i.e., a binary vector of length *c*. A necessary requirement is that different symbols are mapped to different binary vectors. Given a set of symbols *S*, a face constraint is a block $B \subseteq S$ in the partition specifying that the symbols in *B* are to be assigned to one face (or sub-cube) of a binary *c*-dimensional cube, without any other symbol sharing the same face. So, face constraints are generated by step of partition search, *c* is the number of internal binary variables of the net, $|Z|$. Every variable $z \in Z$ corresponds to some two-block partition on *S*. Let the binary internal state variable z^i_j be produced by the sub-machine A_i. Then z^i_j is a state variable of sub-machine A_i and corresponds to the two-block partition h^i_j. One of the blocks of h^i_j is coded by 0, the other one by 1. In this step we decide an combinatorial problem called face hypercube embedding [3], to find the minimum *c* and related $e: S \rightarrow \{0, 1\}^c$ such that face constraints are satisfied i.e., $h_{ij} \leq \boldsymbol{p}_i$.

The internal binary variables for FSM's network are:

$$h_1^1 = \{\overline{1,3,4,5,6}; \overline{2,7,8,9}\}; \quad h^1{}_1 \sim z_1$$

$$h_2^1 = \{\overline{1,3,5,6}; \overline{2,4,7,8,9}\}; \quad h^1{}_2 \sim z_2$$

$$h_1^2 = \{\overline{1,2,4,6,7}; \overline{3,5,8,9}\}; \quad h^2{}_1 \sim z_3$$

$$h_1^2 = \{\overline{1,3,4,7,8}; \overline{2,5,6,9}\}; \quad h^2{}_2 \sim z_4$$

We suppose that the first block of two-block partition corresponds to the value of internal binary variable equal to 1 and the second block corresponds to 0. For instance, if $z_1=1$, than it means that global state s_p the network contains in the first block of partition $h^1{}_1$, *i.e.,* $s_p \in \{1,3,5,6\}$. In this example two-blocks $\mathrm{h}^{\mathrm{i}}{}_{\mathrm{j}}$ partitions were constructing by trivial combining of blocks of corresponded $\boldsymbol{p}_i$ partitions, since the optimal algorithm of search for optimal solution of coding problem is beyond of the bounds of this work.

2.6.3 Determining of the structure of the network

To determine the structure of the network the $M(p_i)$ partitions should be found for each of p_i partitions. In our example:

$$M(p_1) = \{\overline{1,4}; \overline{2,6,7}; \overline{3}; \overline{5,9}; \overline{8}\}$$

So $M(p_1) \leq h^1{}_1 \cdot h^2{}_1 \cdot h^2{}_2$. It means that z_1 is the state variables and z_3, z_4 are the input variables of the first sub-FSM. Table 2.2

B_p	$\alpha_{pq}(B_p, B_q)$	B_q	β_{pq}
a_1	z_3 z_4	a_1	–
	^z_3 z_4 x_4	a_1	y_3
	^z_3 z_4 ^x_4 ^x_5	a_3	y_3 y_4
	^z_3 z_4 ^x_4 x_5	a_2	y_4
	^z_3 ^z_4 x_6	a_1	y_7
	^z_3 ^z_4 ^x_6	a_2	y_7
	z_3 ^z_4	a_2	–
a_2	z_3	a_2	–
	^z_3 z_4 x_5	a_1	y_3 y_4
	^z_3 z_4 ^x_5	a_2	y_4
	^z_3 ^z_4 x_6	a_1	y_7
	^z_3 ^z_4 ^x_6	a_2	y_3 y_4 y_7
a_3	1	a_1	–

Table 2.2 – State Transition Table of constructed component FSM A

The second component sub-FSM can be constructed in the similar way.

2.6.4 Defining of the basis of the network

This step characterizes the basis of realization of the network. The set of states of component FSM A^i is equal to the set of blocks of partition p_i. The internal inputs of component machines are defined at the previous step of procedure. The schematic representation of the constructed network is presented in Figure 2.2. Note that the procedure of construction of network output function g is not provided by this thesis.

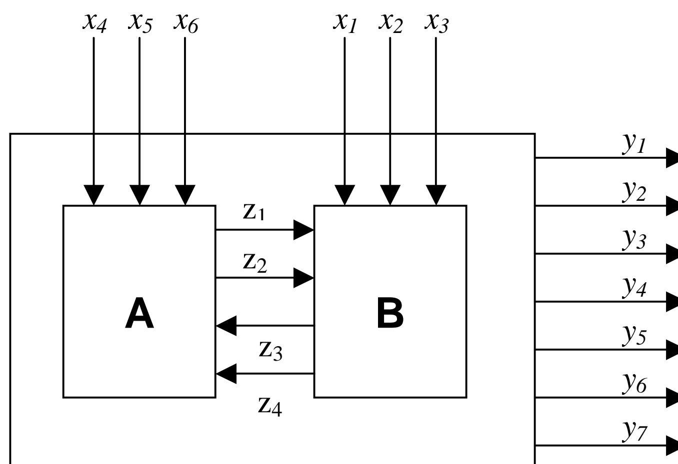

Figure 2.2 – Schematic representation of the constructed network

2.7 Partitions Search Problem

Below we use the representation of Boolean function with complexes of cubes [4]. Two products (cubes) C and C' are in *the relation of consensus* (C *con* C') if and only if they have opposite values (*0* and *1*) exactly in one bound component. Two covers K_1 and K_2 are in consensus if and only if there are $C \in K_1$ and $C'. \in K_2$, which are in consensus.

For every $x \in X$ we define such symmetric binary *relation* ω on S that $s_p \, \omega \, s_q \ (p \neq q)$ if and only if for some s_t exist a-transitions $\langle s_t, s_p, \alpha_{tp} \rangle$ and $\langle s_t, s_q, \alpha_{tq} \rangle$ such that correspondent input conditions α_{tp} and α_{tq} are in consensus. As a result of transitive closure operation of relation ω we will receive *symmetric and transitive relation on S* which we represent as PDC and call it *α-partition* on *S*.

If $x \in X$, than $\alpha(x)$ is PDC on S such, that $s_i \sim s_j \ (\alpha(x))$ means that transitions from state s_i and s_j are the same if input variable x is masked (s_i and s_j are "indistinguishable" by the input channel x).

The notion of relation ω for input variable x_3 for example FSM (Table 2.1) could be presented as the set of following pairs:

$$\{ \langle 3, 5\rangle, \langle 5, 3\rangle, \langle 7, 9\rangle, \langle 9, 7\rangle, \langle 8, 9\rangle, \langle 9, 8\rangle \}$$

The result of transitive closure could be presented as PDC $\alpha(x_3 = \{\overline{3,5}; \overline{7,8,9}\}$. Note that only non-special blocks are explicitly written while block of Don't Care area

(which may not be empty) is omitted. In this case block of Don’t Care area is equal to *{1, 2, 4, 6}*.

For every PDC *r* we put in accordance partition *p Î G(r)*, which defines component FSM A_i in the network *N*. The number of states of component FSM is equal to the number of blocks in corresponding partition *p*.

Affirmation 1 Let *a(X*)* is the sum (the least upper bound) of all *a* -partitions *a*(x_i) such that x_i *Î X** and A_i is a component FSM which is constructed in accordance with some partition from *G(a(X*))* than behaviour of A_i does not depend on all prime inputs of network from *X** if and only if p_i ³ *a(X*)*.

For distribution of output variables among component FSMs we introduce the notion of *b-partition.*

The output associated with a state *s* is the *m*-tuple <y_1, ... , y_m>, where $y_i = 0$ or $y_i = 1$.

For every y_i*Î Y* we define two-block partition on the set of states of FSM *b*(y_i) such that any two states *s* and *t* are in the same block of partition if associated with them output *m*-tuples *l(s)* and *l(t)*have the same *i*-th component, i.e., *s ~ t (b(y))*. The possibility of distribution of output variables is based on the next affirmation.

Affirmation 2. Let *A* is the component FSM associated with partition *r* than the value of binary output variable y_i could be computed by FSM *A* if and only if *r £ b*(y_i).

Distribution of input and output variables can be formulated as a *binate covering* problem and solved exactly or heuristically by the corresponding algorithms [2].

In our approach the heuristic algorithm uses the criteria of informational relationship measure (Chapter 2.5) for constructing the set of partitions for decomposition. The algorithm works in the following manner. The each partition (π_i) of the set is constructed separately, as a sum of the basic α-partitions. The value of informational relationship measure is used on each step of construction of π_i to determine which of the α-partitions will make the product of already constructed partitions be most closer to zero partition.

The work on implementation of algorithm that will include methods for distribution of outputs variables between the component automata is still in progress.

3 Decomposition Software

In this chapter we are going to present the brief overview of the developed software and the detailed guide for using web-based and command-line versions of the programs. The whole software package including ready-to-run GUI and command-line binaries, source code and API documentation can be found on a CD included with this thesis (Appendix C). The web-based part of the software is available at the URL [14].

3.1 Overview

Decomposition and Synthesis software (D&S) is a package of Java programs that is intended to perform decomposition of source FSM and synthesize network of interacting sub-FSMs. The brief list of features provided by the D&S software package includes:

- Decomposition of FSM on specified set of partitions
- FSMs network construction
- Partitions search by specified criteria
- Random generation of set of partitions
- Import and export of FSM/Network (various formats are supported)
- Built-in libraries of FSMs
- Steady-state probabilities computation (by using integrated module of FSMNetSE software [12])

The schematic picture that describes relations between the software components along with workflow process is presented in Figure 3.1.

Since Java technology was used to create the software, the range of platforms and operating systems where the program can be executed is limited only by availability of Java Runtime Environment (JRE) for the concrete system. The D&S software requires Sun-compatible JRE [13] version 1.1.8 or later as a running environment.

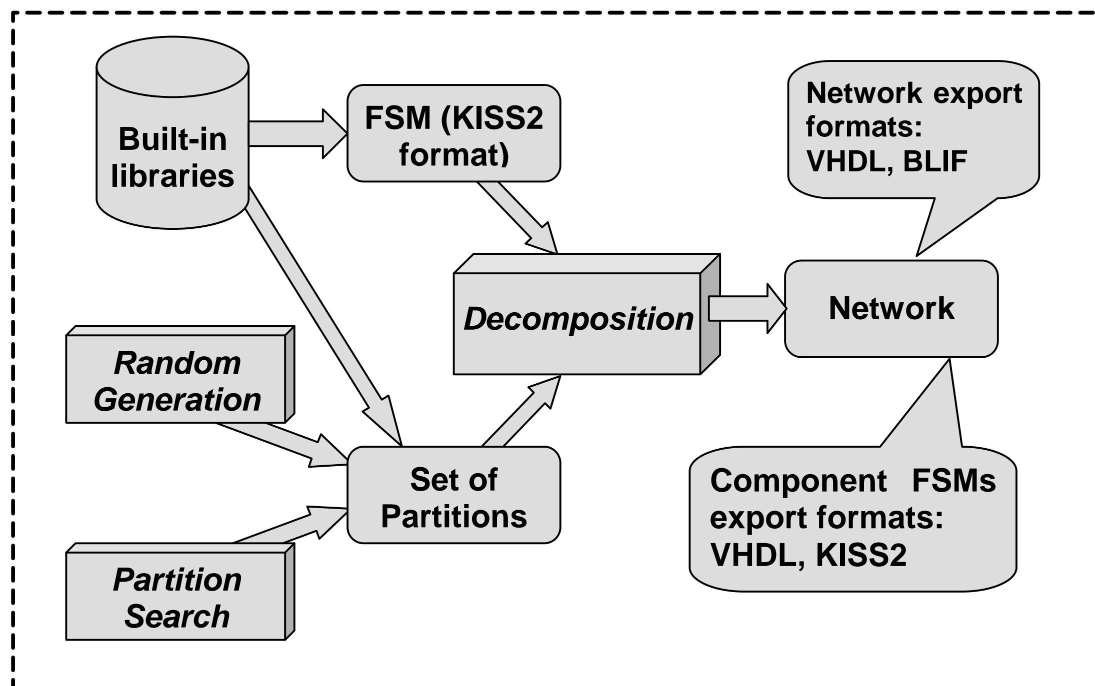

Figure 3.1 – Structure of the software

Most part of the software is implemented in the form of web-based applet with easy-to-use graphical user interface (GUI) and can be accessed over internet. Some components of the GUI program are also duplicated by command-line versions of tools that are useful for carrying out batch experiments.

In the rest of this section we are going to describe in more details both versions of the D&S software package and give an overview of the software API.

3.2 Graphical Interface Tool

The GUI version of the software is implemented in the form of Java applet. The applet can be either accessed remotely via WWW [14] or executed on the user's computer locally. In both cases program requires the Java Runtime Environment (JRE) [13] be properly installed on the client computer. Although the most of implemented features in the applet are available under JRE v.1.1.8, the steady-state probabilities computation requires JRE v.1.3.1 or later. Note that since Microsoft JVM (that is Sun JRE v.1.1.8

compatible) is usually included with the standard installation of Internet Explorer 5 or higher, no additional software is needed to use the most features of the program under Windows environment.

To run the applet remotely user should open the URL [14] in Internet browser with embedded Java support. For starting software locally the ***DSA.html*** file that is located in the applet's root directory (see Appendix C) should be opened in browser.

There also alternative way to start the program without browser on condition that Sun Java Platform is installed on client computer. In this case the applet can be started by following command:

```
appletviewer <URL>
```

where *URL* is the address of web page where the applet is situated or local ***DSA.html*** file.

3.2.1 Interface description

The working window of the applet (Figure 3.2) consist of menu bar, status bar and three main tab-panels: "*Decomposition*", "*Network*" and "*Components*". The "*Decomposition*" tab-panel is enabled by default while the last two panels are available only after decomposition of FSM was performed successfully.

The status bar (that is located in the bottom area of the working window) is divided into two parts. The state of the program that is indicated in left side of the bar is supplemented with detailed description of the status in right part. The state of the program could be one of the followings:

- *Ready* – the program is currently ready to start the decomposition process
- *Working* – one of the computation processes (decomposition, partition search, etc) is currently executing
- *Loading* – the program is at the initialization stage
- *Decomposed* – decomposition was finished successfully and network of FSMs was constructed
- *Error* – decomposition cannot be performed because input data are invalid or absent

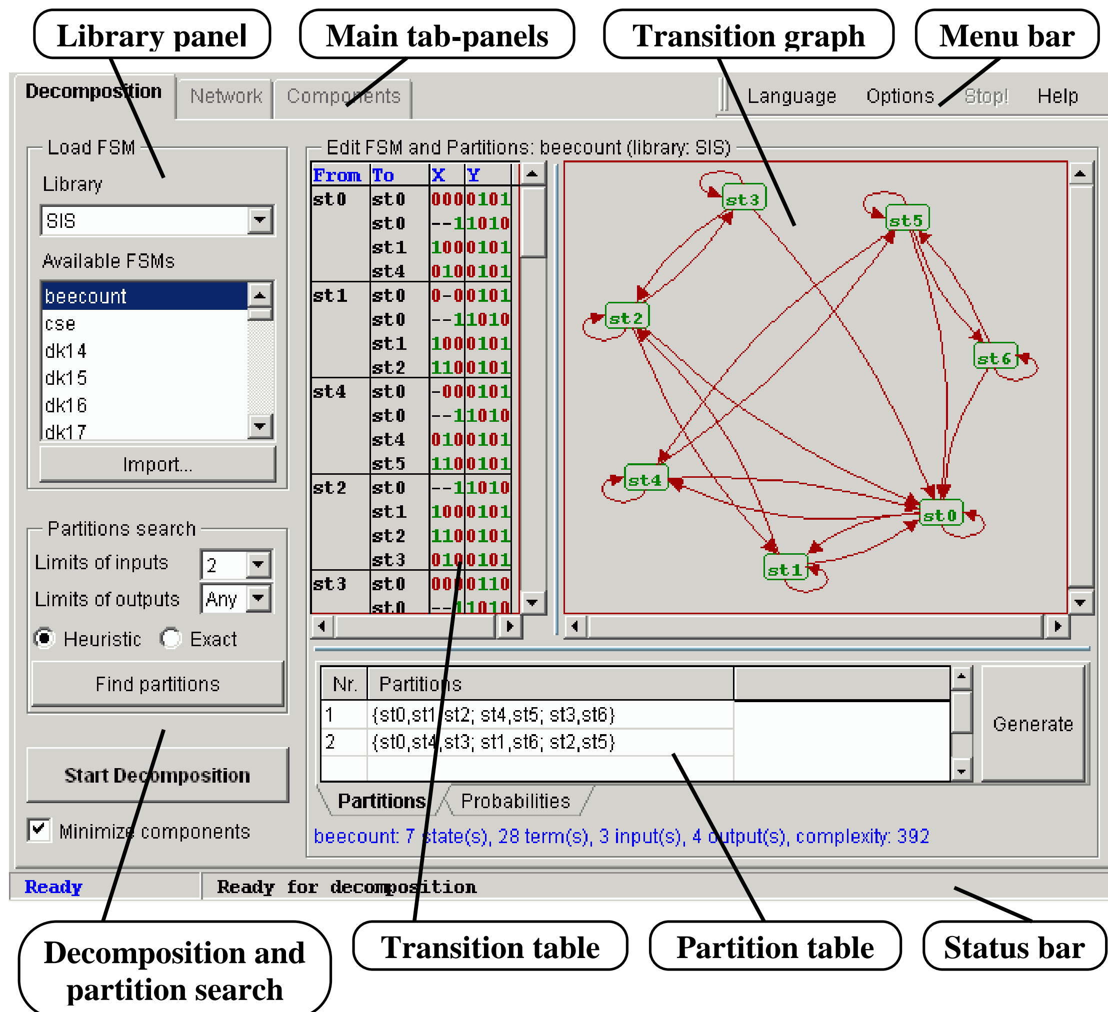

Figure 3.2 – Main window of the applet

The menu bar of the program consist of the following commands:

- *Language* – allows to select the language of the interface from the list of supported languages. The detailed description of configuration files for multilingual support is presented in Appendix A.
- *Options* – allows user to adjust several parameters of the user interface
- *Stop!* – this command becomes enabled during computation processes. Clicking this command while program is in the "*Working*" state interrupts active process before finishing
- *Help* – brings out the "*About*" dialog with name, version and other information about the program.

The basic steps of work with the GUI version of the software are described below.

3.2.2 Loading source FSM

Before starting to work with the program source FSM for experimenting should be loaded using the "*Load FSM*" sub-panel (Figure 3.3). There are two possibilities to do this: select FSM from built-in library or load it from external description.

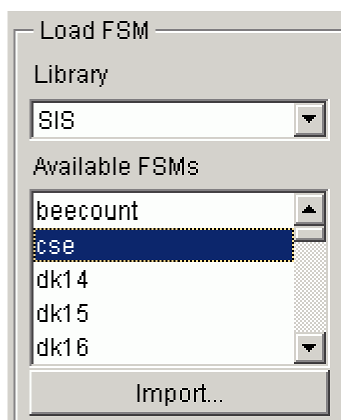

Figure 3.3 – Load FSM panel

The contents of the specific library that can be selected using the "Library" list is displayed in the "*Available FSMs*" box. Another opportunity to specify FSM is to provide program with its description in the Berkeley KISS2 format (for more information about formats see Appendix B) using the "*Import…*" command. In case (if supplied description is correct) the FSM will be loaded into the applet and its name will appear in the contents of *User* library.

The state transition graph and state transition table of the main panel (Figure 3.2) represent currently loaded FSM in the corresponded form and allow to edit its structure.

3.2.3 Specifying set of partitions

The applet checks automatically if specified set of partitions (Figure 3.4) is valid (partitions have correct format, product of partitions is equal to zero partition, etc) and enables "*Start Decomposition*" button. In case if entered set of partitions does not meet some of the criteria, the invalid partitions will be marked by red color and appropriate error description will be printed in the status bar. Below, detailed description of several methods for obtaining partitions for decomposition is presented.

Partition search

The program is capable to search set of partitions for decomposition that will satisfy the criteria of distribution of primary inputs. This process is controlled by "*Partition*

Search" sub-panel. The "*Limit of inputs*" defines the maximum limit of primary inputs that each of component automata will depend on. If user selects "*Min*" value the applet will try to find the solution with minimal possible number of primary inputs in each of sub-FSMs.

There are two algorithms implemented for solving the search problem: heuristic and exact. The first one uses a heuristic technique (Chapter 2.7) to find the set of partitions and therefore does not guarantee that the solution will be found. Another algorithm uses an exhaustive search and always finds the optimal solution. However, the last method usually requires a lot of processor time, thus it is reasonable to use it only for FSMs with relatively small number of primary inputs.

Random generation

Clicking the "*Generate*" button in the "*Partitions*" sub-panel (Figure 3.4) will force the applet to randomly generate set of partitions for decomposition. Note that decomposition on randomly generated set of partitions may result in construction of network with large complexity and tangled structure, therefore several attempts are usually needed to obtain the acceptable results.

Manual specification

The set of partitions can be entered manually using table the "*Partitions*" sub-panel (Figure 3.4). Clicking the left mouse button on existing partition or on empty row allows to enter new partition in the table, while clicking the right mouse button brings out the context menu.

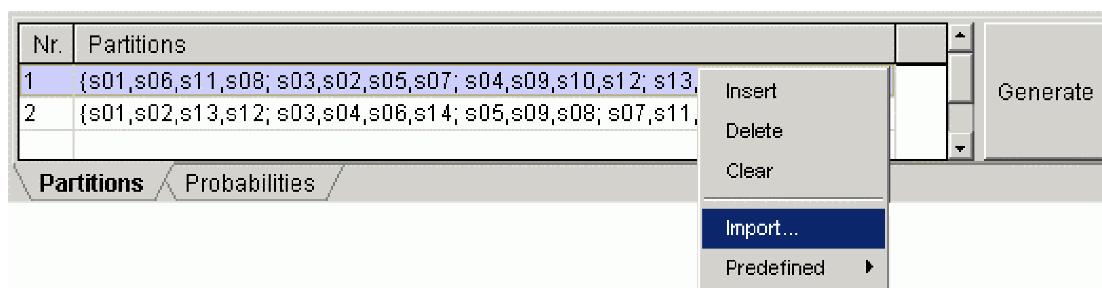

Figure 3.4 – Partitions table panel

The whole set of partitions could be also loaded from the file by using the "*Import...*" command from the context menu. The format of partitions is described in Appendix B.

Preset partitions

The list of preset partitions in built-in library (Chapter 3.2.6) associated with currently loaded FSM is available under "*Predefined...*" item of the context menu. By default first item in the list is selected after the loading of FSM.

3.2.4 Decomposition

The procedure of decomposition can be started by clicking the "*Start Decomposition*" button that becomes enabled only if the specified input data are valid.

Un-checking the "*Minimize components*" checkbox (that is placed into "*on*" state by default) can greatly speed up the whole process of decomposition in certain cases, by omitting the phase of minimization of component automata. In this case components will be left un-minimized that can affect on the estimation of the complexity of obtained network. However, the network could be minimized later using special software for Boolean minimization (for example [7], [8]).

The results of decomposition are the constructed network (that is displayed in the "*Network*" tab-panel) and the list of component automata (the "*Components*" tab-panel). The sample of constructed network is presented in Figure 3.5. Each of the rectangles represents a component sub-FSM in the network, while the primary inputs/outputs of the network as well as internal interconnections are shown by the colored lines.

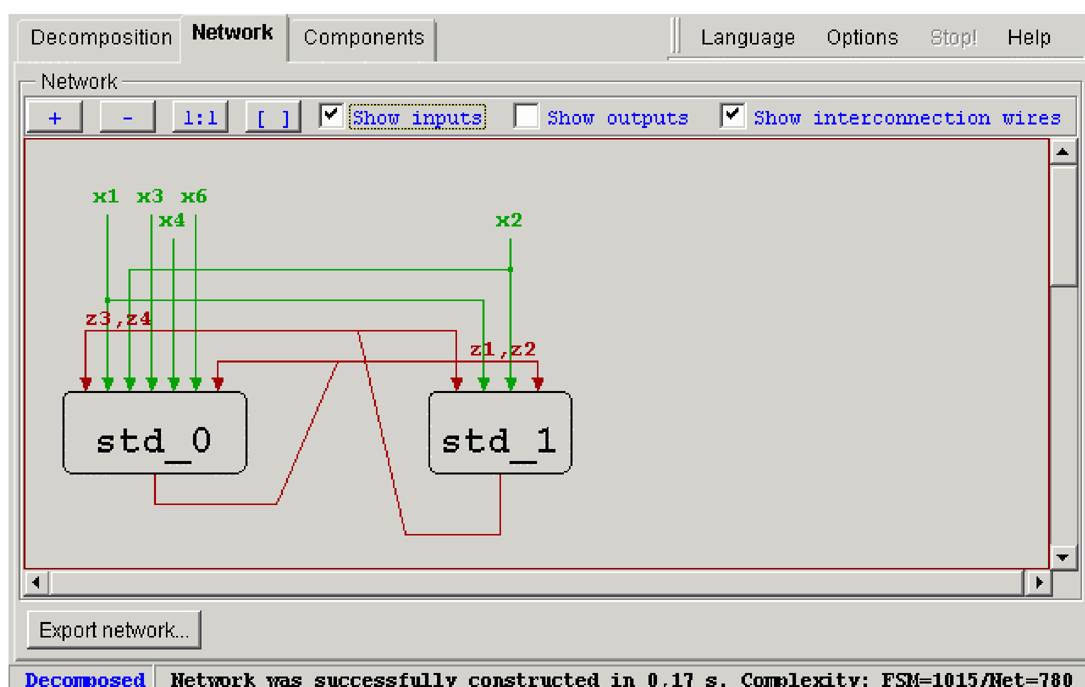

Figure 3.5 – Panel with constructed network

The structure of result network can be exported for further exploration (the *"Export Network"* button) to another CAD-related software. The program support such widely used formats as VHDL and BLIF. Also all the components of network can be exported one by one in KISS format.

The components of the network are presented in the "*Components*" tab-panel (Figure 3.6). Selection sub-FSM from the list will bring corresponded transition table and transition graph to the user's screen. The "*Remove unused I/O*" checkbox controls whether unused inputs and outputs lines should be removed from the FSM description or not. Another checkbox "*Coded Z inputs*" set that z-signals be displayed by binary or symbolic codes.

Figure 3.6 – Panel with FSM's network components

The bottom tab-panels provide user with different information about the constructed network such as computed M-operator, coding partitions, network states assignments, etc. Each of component sub-FSMs of network can be exported into Berkeley KISS2 format description using the "*Export…*" command.

3.2.5 Probabilities computing

The GUI version of the program also supports computation of steady-state probabilities of FSM using the API of FSMNet Stochastic Explorer Software [12].

The screen-shot of “*Probabilities*” tab-panel is presented in Figure 3.7. The left table contains source input patterns and probabilities of appearing of corresponded patterns on the input of FSM. These probabilities can be either generated (“*Generate…*” command of the context menu of input patterns table) or specified manually. If the input probabilities is not specified anyhow equiprobable input patterns will be used.

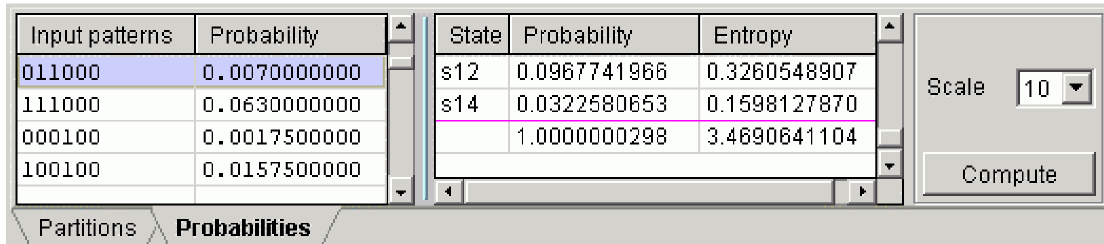

Figure 3.7 – Probabilities tab-panel

The “*Scale*” option determines accuracy (number of digits after the decimal point) of the computation. More information about the steady-state probabilities computation can be found in [12].

3.2.6 Built-in libraries

Built-in libraries represent a grouped collection of FSMs and partition sets that helps user to avoid entering frequently used input data manually. The program stores set of libraries, each of which is capable to contain a collection of FSMs. In addition, several sets of partitions can be associated with every stored FSM. This mechanism allows user to easily search and load source data for the experiments. The library of benchmarks [9] is included with the software by default.

Although, there no possibility to modify contents of built-in libraries by means of the current version of the software, there is still possibility to do this by placing appropriate files into special directory of the program (see Appendix A for details).

3.3 Command-line Tools

The command-line version of decomposition software can be more useful for carrying out experiments on set of benchmarks (for example [9]). It uses less system resources, can be executed on character terminal of server and allows to read/write user files.

Currently four basic commands are supported: *decomposition*, *partition search*, *random partition generation* and *checking the FSM specification*.

The command-line tools can be executed only locally via the corresponded shell-script file (***ds.bat*** under Windows or ***ds.sh*** in case of Unix system). The Java classes that implement all the functionality are placed into the ***cmd.jar*** file.

Before the first use of command-line version of the software be sure that the following requirements are fulfilled:

§ Java interpreter is listed in the system path variable or its location is strictly specified inside the shell-script file.

§ The *CLASSPATH* variable inside the shell-script (or in system shell environment) points to the right location of ***cmd.jar*** file.

The command-line format under Unix environment is:

```
ds.sh <command> <arguments>
```

In case of Windows-based system:

```
ds <command> <argument>
```

The detailed description of the each of commands is provided below.

3.3.1 Decomposition

The decomposition of specified FSM on given set of partitions can be performed by the command ***Decomp***. The command has the following format:

```
Decomp [options] <FSM> <Partitions>
```

By default the ***Decomp*** command decomposes given FSM into the network, performs coding of the network and minimization of component automata and prints complexity of obtained network in comparison with source FSM to the screen. The behavior of the command can be modified using one or more optional switches. The detailed description of the parameters and switches is given in Table 3.1.

Parameter	Brief description
<FSM>	The name of file that contains the input Finite State Machine specification in Berkeley KISS2 Format.
<Partitions>	The name of file with set of partitions that will be used to perform the decomposition.
Option	**Brief description**
-file <name>	Specifies name of the file where output network should be stored.
-blif	Indicates that output network should be stored in *Berkeley Logic Interchange Format* (BLIF). Cannot be specified with **-nc** option.
-vhdl	Indicates that output network should be stored in VHDL format. The format of VHDL file depends on coding of the network during the decomposition process. If the coding of the network is omitted (by "***-nc***" option) the behavioral VHDL description will be generated. The states of the component automata will be specified by symbolic names instead of codes. In case if coding is performed the output VHDL file will contain structural description of the network (this option is not fully supported by the current version)
-nc	Specifies to not perform coding of network states
-nm	Specifies to not perform minimization of Boolean functions of component automata.
-v	Prints additional information about the decomposition process. The scheme of relations of components in network, M-partitions and information about network states coding will be printed.
-debug	Prints useful debug information to the screen

Table 3.1 – Parameters and options of the *Decomp* command

If no parameters are specified for this command the help message with usage options will be printed on the screen.

If the output format is omitted the results of decomposition will be stored in default output format. This means that the output file will contain a list of component automata in KISS2 format and each of the components will be preceded by a commented section with additional information about the structure of network. The examples of other output formats are presented in Appendix B.

Since the minimization algorithm is not very powerful, the minimization can require a lot of processor time in certain cases. Therefore sometimes the "***–nm***" option can greatly speed up the execution of command.

A screen-shot of execution of the ***Decomp*** command under Windows environment is given in Figure 3.8.

```
C:\WORK\test\cmd>java Decomp std.kiss2 std.part

Loading FSM [std.kiss2] OK
Loading partitions [std.part] OK

Starting decomposition

Phase 1: decomposition
Phase 2: coding/minimization

Decomposed successfully in 0.11 s
Net complexity: 729, original FSM complexity: 1015  (28.17%)
```

Figure 3.8 – Sample execution of the *Decomp* command

3.3.2 Partition search

The ***Search*** command is intended for generating set of partitions that can be used for FSM decomposition. The theoretical basis of the procedure of constructing partitions for decomposition is described in Chapter 2.7.

There are two algorithms implemented for this purpose. One of them uses a heuristic technique to find the set of partitions, but does not guarantee that this set can be found. On the other hand, another algorithm uses an exhaustive search and always finds the optimal solution. Although this algorithm usually requires a lot of processor time, it can be still used with FSMs that have relatively small number of primary inputs.

Below the format of this command is presented:

```
Search [options] <FSM>
```

The parameters and options of the command is described in Table 3.2.

Parameter	Brief description
<FSM>	The name of file that contains the input Finite State Machine specification in Berkeley KISS2 format.
Option	**Brief description**
-file <name>	Specifies name of the file where to write found partitions. If this option is omitted found partitions will be printed on the screen.
-h	A heuristic algorithm will be used for partition search (this mode is default).
-e	Will use exhaustive algorithm for searching partitions. The algorithm can take a very long period of time to execute.
-a	Only construct α-partitions and exit.
-i <num>	Specifies limit of primary inputs in the component automata. If this option is not specified the algorithm will try to find partition to satisfy minimal possible limit of inputs.
-v	Prints additional information about the decomposition process. The α-partitions and their cover will be printed on screen.
-debug	Prints useful debug information to the screen.

Table 3.2 – Parameters and options of the *Search* command

If no parameters are specified for this command the help message describing the usage options will be printed on the screen. A sample screen-shot of execution of the command under Windows environment is presented in Figure 3.9

```
C:\WORK\test\cmd>java Search std.kiss2
Starting search [heuristic algorithm]

{s01,s02,s05,s13; s03,s04,s06; s07,s09,s08; s11; s10; s12; s14}
{s01,s03,s07,s11,s10,s12,s14; s02,s06,s09; s05,s04,s08; s13}

Partitions for decomposition found successfully (inputs limit: 4)
Time elapsed: 0.06 s
```

Figure 3.9 – Sample execution of the *Search* command

3.3.3 Generation of random set of partitions

This command allows to experiment with randomly generated partitions. The command ***Random*** is capable to generate standalone partitions and set of partitions for decomposition. It is also possible to estimate quality of the generated partition sets and select the best one. The command ***Random*** has the following format:

```
Random [options]  <FSM or power>
```

The detailed description of the parameters and options of the command in provided in Table 3.3.

Parameter	Brief description
<power>	Number of elements in target partition(s), must be a positive integer value
<FSM>	The name of file that contains the input Finite State Machine specification in Berkeley KISS2 format. The number of states in the FSM will be used as a power of partition(s)
Option	**Brief description**
-file <name>	Specifies name of the file where found partitions should be written. If this option is omitted partitions will be printed on the screen.
-seed <num>	Sets initial seed of the random generator. Must be an integer value between *0* and $2^{31}-1$. If this option is not specified the seed will be initialized to value based on current time. The execution of the command with equal seed values will generate equal sequences of partitions.
-blocks <num>	Specifies number of blocks in target partition. If omitted, each of generated partitions will consist of random number of blocks.
-count <num>	Number of partitions/partition sets to generate. Default is *1*.
-z	Indicates that partition set(s) should be generated instead of standalone partitions. The product of multiplication of all partitions in the set will be *zero-partition*.
-best	The same as **-z** option but also performs an estimation of quality of the each of generated partition sets. The only partition set that provides best (minimal) decomposition of FSM will be stored as the result of execution of the command. If this option is specified a FSM must be provided as the parameter of this command.
-nm	This option can be only used with **-best** option. If specified, minimization of the component automata of network during the estimation process will not be performed. This option can greatly speed-up the execution of the command but also decreases the accuracy of estimation.

Table 3.3 – Parameters and options of the *Random* command

The generated partition(s) will contain all the numbers between *0* and *specified power-1*. In Figure 3.10 is presented a screen-shot of sample execution of this command under Windows environment. In this example 100 partition sets for decomposition of given FSM (***std.kiss2*** file) were randomly generated. The best result of decomposition (reduction of complexity by 3.74 percents) was obtained using 21st of generated partition sets. This set of partition was printed on the screen.

```
C:\WORK\test\cmd2>java Random -best -count 100 std.kiss2
{s01,s12; s03; s02,s10; s05; s04; s13; s07; s06; s09,s14; s11,s08}
{s01,s02,s04,s08,s14; s03,s05,s13,s07,s06,s09,s11,s10,s12}

100 partitions sets generated in 2.254 s (44.3/s)
Best result [21]: 977/1015  (3.74%)
```

Figure 3.10 – Sample execution of the *Random* command

3.3.4 Checking format of the FSM

The command ***Check*** verifies the format of supplied FSM description and determines whether FSM is deterministic and completely specified or not. The command also prints detailed information in case of an error in the format of FSM description. If given FSM contains conflict transitions (different transitions from one state under the same input conditions) or it is not completely specified (don't care input combination exist for some state) the corresponded information will also be printed on the screen. The command has the following format:

```
Check [options] <FSM>
```

The description of the parameters and options of the command is provided in Table 3.4.

Parameter	**Brief description**
<file>	The name of file that contains the input Finite State Machine specification in Berkeley KISS2 format.
Option	**Brief description**
-c	Do not check FSM for determinism
-d	Do not test if FSM is completely specified or not
-q	Stops execution on first found conflict transition or incompletely specified state of FSM

Table 3.4 – Parameters and options of the *Check* command

3.4 Software API

In this chapter we are going to present a brief description of the Application Programming Interface (API) of the software. On the reasons of contracting the number of pages, the chapter contains only the overview description of the main parts of the software core. The whole documentation of the API along with source code is provided on a CD that can be found in Appendix C.

3.4.1 API Overview

The D&S API provides a developer with the powerful framework that can be used for creating new applications on decomposition or adding functionality to already developed software. The API is written on Java and is compliant to Sun Java 1.1 specification.

Below the essential core classes of the API (that deal with decomposition algorithm, storage and processing of FSMs, networks, partitions, etc) are described. The whole documentation of source classes is included on a CD (Appendix C).

3.4.2 The API Core Classes

The classes of ***fsm.matrix*** package are intended to implement *tri-state vector* and *tri-state matrix* data structures. These classes are used to represent input conditions and output signals in transition table of FSM.

Vector implements a *tri-state vector* (the elements of vector can be placed into *High*, *Low* or *Don't Care* states)

Matrix implements a matrix that consist of tri-state vectors. The class provides methods for manipulating with such matrices, testing matrices for logical equivalence, etc.

Optimizer realizes the algorithm of minimization of Boolean functions that are represented by tri-state matrices.

OrthogonalVector implements the algorithm of search for the tri-state vector that will be orthogonal to all the rows of tri-state matrix. This algorithm can be used in many tasks of logical synthesis

The classes of ***fsm.partition*** package implement set, partition and PDC data structures. The package also contains implementations of various algorithms for solving the partitions search problem (Chapter 2.7) and random partition generation.

AlphaBuilder used to find set of α-partitions for given FSM

ExhaustiveSearch realizes the algorithm of exhaustive search of partitions set for decomposition

HeuristicSearch heuristically finds set of partitions for decomposition

Partition provides useful methods for storing partitions and manipulating with them. Realizes various operations on partitions (adding, multiplication, etc)

PDC this class is inherited from ***Partition*** class in order to support the notation of Partition with Don't Care's.

PartitionSearch realizes the full procedure of search of partitions for decomposition

RandomGenerator implements methods for random generation of partitions and partitions sets

Set implements a set of elements. Contains methods that realize various operations on sets (adding, multiplying, etc)

The ***fsm*** package contains classes that represent FSM/Network and perform the procedure of decomposition

FSM implements a Finite State Machine. The various types of FSMs (Mealy, Moore, etc) are supported. This class contains all the needed methods for manipulation with stored State Transition Table of FSM. The internal information about STT is stored by a number of ***State*** classes. The ***FSMType*** and ***FSMInfo*** classes are used to store internal information about type (Mealy, Moore, etc) and basic parameters (number of inputs, outputs, etc) of FSM correspondingly.

State represent description of a state of FSM

Term represent a term of FSM transition table. Term consist of information about present and next states, condition of transition from present to next state and corresponded output signals

Network stores a network at intermediate stage of decomposition (network states are still not coded)

CodedNetwork stores a final network where all states are appropriately coded

Decomposition performs decomposition of given FSM into network of interacting automata. Stores the results using the instance of ***Network*** class

Coding performs the task of coding of states of network. The supplied network should be represented by ***Network*** object and the final decomposition results are stored by the instance of ***CodedNetwork*** class

Format handles with export and import of FSM/Network from/to various formats. At this moment KISS2, BLIF and VHDL formats are supported by this class.

The classes of ***fsm.prob*** package are used as the wrappers to the interface of the FSMNetSE API (that realizes the computation of steady-state probabilities of FSM).

4 Conclusions

4.1 Thesis Summary

This thesis presents a newly developed system that provides user with the essential software environment for investigations in the area of FSMs decomposition. The developed software implements the original decomposition approach that was investigated at Department of Computer Engineering of Tallinn University of Technology. The system is intended to be used for two basic purposes.

First of all, it can act as a research instrument for investigations on decomposition. In addition to web-based GUI version of software that gives the possibility to execute experiments interactively, the included command-line tools are useful for carrying out benchmarks. Due built-in support of widespread data formats, the experiments can be carried out on the range of well-known benchmarks of FSMs as well as on used-defined FSMs. Moreover, the results of experiments can be exported to other CAD-related software for further investigations.

Besides this, the developed system can be easily used for educational purposes, because of it interactive nature and user-friendly graphical interface. In this case, the great advantage is that the system can be easily accessed from any point of the world over Internet.

The decomposition API was also developed as a part of the work on the system. This feature provides developers with the framework for extending functionality of currently developed programs and creating new software in this area.

4.2 Future Work

Future work on the system can be performed in the following directions. More efficient algorithm of encoding of network states (Chapter 2.6.2) should be implemented in order to improve the overall efficiency of decomposition. As a partial solution of this problem, the capability to use results of state assignment obtained by other software can be built into the system.

A little work has to be done in order to implement the support of structural description of FSMs network in VHDL format.

The improvements can be also made in the algorithm of partition search (Chapter 2.7) in order to allow distribution of FSM outputs over sub-machines (in addition to distribution of inputs that is currently implemented).

5 References

[1] Hartmanis., J., Stearns, R.E. *Algebraic Structure Theory of Sequential Machines*. Englewood Cliffs, N. J.: Prentice-Hall, 1966

[2] Devadas, D. *General Decomposition of Sequential Machines: Relationship to State Assignment* // Design Automation Conference (DAC'89), pp. 13-27, 1989.

[3] *Theory and Algorithms for Face Hypercube Embedding* / Goldberg, E., Villa, T., Brayton, R., Sangiovanni-Vincentelli, A. // IEEE Transactions on Computer-Aided Design, vol. 17, no 6, pp. 472-488, 1998.

[4] De Micheli, G. *Synthesis and Optimization of Digital Circuits*, McGraw-Hill, 1994.

[5] Jozwiak, L., Kolsteren, J. *An Efficient Method for the Sequential General Decomposition of Sequential Machines* // Microprocessing and Microprogramming, North Holland, vol. 32, pp. 657-664, 1991.

[6] *SIS: A System For Sequential Circuit Synthesis* / Sentovich, E., Singh, K., Lavagno, L., Moon C., Murgai, R., Saldanha, A., Savoj, H., Stephan, P., Brayton, R., Sangiovanni-Vincentelli, A. Department of Electrical Engineering and Computer Science, University of California, Berkeley, 1992.

[7] Rudel, R. *Espresso – Boolean Minimization*. Department of Electrical Engineering and Computer Science, University of California, Berkeley, 1992

[8] Rudel, R., Sangiovanni-Vincentelli, A. *Exact Minimization of Multiple-Valued Functions for PLA Optimization* // IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. CAD-6, no. 5,. p. 727-50, 1987.

[9] McElvain, K. *LGSynth'93 Benchmark Set: Version 4.0*, 1993. [WWW] http://www.cbl.ncsu.edu/benchmarks.

[10] Sudnitson, A. *Computational Kernel Extraction for Synthesis of Power-Managed Sequential Components* // IEEE 9th International Conference on Electronics, Circuits and Systems (ICECS 2002), Dubrovnik, Croatia, 2002, pp.749-752

[11] Sudnitson, A. *An Approach to Synthesis of Mixed Synchronous/Asynchronous Digital Devices* // IEEE 23rd International Conference on Microelectronics (MIEL 2002), Yugoslavia, 2002, pp. 695-698

[12] Meškov, M. *Stochastic Analysis of Finite State Machine Networks*. Tallinn University of Technology, Tallinn, 2002.

[13] Java 2 Platform Standard Edition, version 1.3.1
[WWW] http://java.sun.com/j2se/1.3/

[14] Web-Based Environment for Decomposition
[WWW] www.pld.ttu.ee/dildis/automata/applets/dsa

Appendix A Configuration Files Format

This section contains the description of the configuration files for the GUI version of decomposition software.

A.1 Library files

The format of files that store contents of the built-in libraries is described below.

All the library related files (except the description of FSMs in KISS2 format) have similar formatting rules. First of all, these files have textual, line-based format. The semicolon (';') is considered as a comment character and the part of line after it (including the comment character itself) is ignored. Empty lines and lines that consist only of comments are treated as non-significant.

The list and the contents of built-in libraries is stored under "***Libs/***" directory of the root directory of the applet. This directory should contain special file with name "***libs***" that stores the list of libraries in the following format:

```
<Lib1_Name> <Lib1_Dir>
[<Lib2_Name> <Lib2_Dir>]
. . . .
```

where *Lib_Name* is the name of the library and *Lib_Dir* is the subdirectory where all the library files should be placed. The contents of each of libraries is specified by the "***list"*** file inside the library's subdirectory. The format of the file is described below:

```
<FSM_name>=<file>,[partitions_file],[comment]
```

where:

FSM_name – string that will be displayed as the name of FSM

file – name of the file that contains description of FSM in KISS2 format.

partitions_file – name of optional file that contains a collection of partitions that can be used for decomposition of this FSM. The format of this file is described below.

comment – optional text comment associated with this FSM. The comment should not contain coma (‘,’) symbols.

The file of partitions should have the following format:

```
<Partitions_set1_name>
[Partition1_1]
[Partition1_2]
. . .
[<Partitions_set2_name>]
. . .
```

where *Partition_set_name* – defines the name of set and *Partition* statement is the partition defined by string in corresponded format.

A.2 Multilingual support

Although program uses English-based user interface by default, there is the possibility to translate interface into any other languages. All the files that provide multilingual support are placed into ***Languages/*** subdirectory of the root directory of the applet:

BundlesDescription – stores the list of currently supported languages along with locations of ****.bdl*** files

****.bdl*** – contain textual strings translated to corresponded language

The ***BundlesDescription*** file has the following format:

```
supportedLanguages=<language1>[,<language2>][,<language3>]
DefaultLanguage=<default>
<language1>=<file1>.bdl
[<language2>=<file2>.bdl]
. . . . .
```

where:

language1, *language2*, …– names of the languages, that will be displayed under the “*Language*” command of the program

default – name of the language that will be used by default

file1, *file2* –names of the files (in the ***Languages/*** subdirectory) that contain the translations

The format of the ***.bdl*** files is the following:

```
<string_id>=<translated string>
```

where:

string_id – identifier of the string (the identifiers should be identical in all the ***.bdl*** files)

translated_string – translation of corresponded string to another language

For translation of program's interface the file ***Empty.bdl***, that located in the ***Languages/*** subdirectory, can be used as base file that contain all the possible string identifiers without translation.

Appendix B   Data Formats

The section contains the descriptions of various data formats that are used by the developed software.

B.1 KISS2 Format

KISS2 is a widely used line-based textual format for description of Finite State Machines that was developed at Berkeley University.

The format requires lines of data to be terminated by one of the following characters: '\r', '\n' or '\r\n'. The part of line after the '#' character is treated as a comment and ignored. The description of FSM in KISS format consist of two parts: header and State Transition Table (STT). The header describes general properties of FSM and is placed before the STT description. It consist of attribute lines that begin with '.' character. The list of possible attributes is provided below.

.i <num> - specifies the number of inputs in FSM

.o <num> - specifies the number of outputs in FSM (can be 0)

.s <num> - (can be omitted) specifies the number of states in FSM

.p <num> - (can be omitted) specifies the number of terms (description lines in the FSM description body)

STT description consist of a number of lines of the following format:

```
<input> <current_state_name> <next_state_name> [output]
```

State names are specified by arbitrary strings that should not contain spaces. Inputs and outputs are specified as *tri-state vectors*. If the number of outputs in header is 0 the outputs should not be specified.

Tri-state vector is a string of the following characters: '*0*', '*1*' and '-' (low value of signal, high value of signal and don't care correspondingly). The length of input/output vectors (that is determined by number of characters in string) should match to the

number of inputs/outputs specified in header. The end of FSM description could be terminated by special line: "`.e`". Otherwise the end is determined by the end of file.

A Don't Care state (that means that no matter in what state the FSM actually is) can be also specified by the KISS2 description in the following manner:

- the Don't Care state should be named by asterisk character ('*')
- the number of states should be specified in the header and this number should not include the Don't Care state

The FSM descriptions containing the Don't Care state are processed in such way, that all the transitions from Don't Care state will be added to all other states of the FSM.

The small example of KISS2 description that represents FSM presented in Chapter 2.6 is provided below:

```
.i 6
.o 7
.s 9
10---- s1 s1 11-----
00---- s1 s3 -1-----
01---- s1 s5 ----1--
11---- s1 s6 1------
-1---- s2 s2 -1---1-
-0---- s2 s7 ----1--
---1-- s3 s1 --1----
---00- s3 s4 --11---
---01- s3 s7 ---1---
--1--- s4 s3 -1--11-
--0--- s4 s5 ----1--
-----1 s5 s5 ------1
-----0 s5 s9 ------1
1-1--- s6 s7 ----11-
0-1--- s6 s8 ----1--
--0--- s6 s9 -----1-
-1---- s7 s8 -1-----
-0---- s7 s9 1----1-
----1- s8 s3 --11---
----0- s8 s8 ---1---
-----1 s9 s1 ------1
-----0 s9 s7 --11--1
```

For more information about Berkeley FSM description format, see *Appendix B* of the *Documentation on SIS* [6]

B.2 Partition Format

This format is used to represent partitions on the set of states of FSM. The partition should be described by separate line that has the following format:

{<block1_state1>[[, <block1_state2>], …, block1_stateN] [; <block2_state1> …] …}

The example of partitions (that were used for decomposition in Chapter 2.6) is presented below:

```
{s1,s3,s5,s6; s2,s7,s8,s9; s4}
{s1,s4,s7; s2,s6; s3,s8; s5,s9}
```

Here, the first partition consist of two blocks. Elements *s1*, *s3*, *s5* and *s6* belong to the first block of partition, elements *s2*, *s7*, *s8* and *s9* form the second block and the element s4 is placed to the last block of partition.

Note that if this format is used for description of PDCs, the special block of Don't Care area should not be written explicitly (all the elements that were not shown in PDC are treated as belonging to Don't Care area of partition).

B.3 BLIF Format

The BLIF (Berkeley Interchange Logic Format) is a well-known format that is used to describe logic-level circuits in textual form. Both combinational and sequential circuits are supported by the format. Additional information about BLIF description can be obtained in the *Documentation on SIS* [6]

In this decomposition software the BLIF format is used for description of constructed network of component automata. The example of network (which was constructed in Chapter 2.6) in BLIF format is presented below.

```
.model Net_ex

.inputs x1 x2 x3 x4 x5 x6
.outputs

.latch  l1_0 z1 0
.latch  l1_1 z2 0
.latch  l2_0 z3 0
.latch  l2_1 z4 0

.names x4 x5 x6 z3 z4 l1_0
--100 1
---11 1
1--1- 1
00-10 1
```

```
--100 1
-1-10 1
---11 1

.names z1 x4 x5 x6 z3 z4 l1_1
1--100 1
1---11 1
11--1- 1
101-10 1
1---01 1
1--00- 1
1--100 1
1-1-10 1
1----1 1
1-0-1- 1
1--00- 1
0---11 1

.names z4 x1 x2 x3 z1 z2 l2_0
110-11 1
100-11 1
1-1-01 1
1--110 1
01-111 1
0-0-01 1
00-111 1
1---11 1
1---01 1
0---01 1

.names z3 x1 x2 x3 z1 z2 l2_1
110-11 1
111-11 1
11-111 1
1-0-01 1
1-1-01 1
0---11 1
0---01 1

.end
```

B.4 VHDL Format

The VHDL is used for representing network of component automata. At this time only behavioral VHDL descriptions are supported by the software. The example of network (which construction was described in Chapter 2.6) in VHDL format is presented below.

```
library IEEE;
use IEEE.std_logic_1164.all;

entity NET_EX is
   port (
      CLK: in STD_LOGIC;
      X: in STD_LOGIC_VECTOR (5 downto 0)    -- primary inputs
   );
end;
```

```
architecture BEHAVIOR of NET_EX is
   type EX_0_STATE_TYPE is (st0, st1, st2);
   type EX_1_STATE_TYPE is (st0, st1, st2, st3);
   signal EX_0_STATE, EX_0_NEXT_STATE : EX_0_STATE_TYPE;
   signal EX_1_STATE, EX_1_NEXT_STATE : EX_1_STATE_TYPE;
begin

   -- Process to hold combinational logic of EX_0
   EX_0_COMBIN: process (EX_1_STATE, EX_0_STATE, X)
   begin
      case EX_0_STATE is
         when st0 =>
            if X(0) = '1' and X(1) = '0' and EX_1_STATE = st0 then
               EX_0_NEXT_STATE <= st0;
            elsif X(0) = '0' and X(1) = '0' and EX_1_STATE = st0 then
               EX_0_NEXT_STATE <= st0;
            elsif X(0) = '0' and X(1) = '1' and EX_1_STATE = st0 then
               EX_0_NEXT_STATE <= st0;
            elsif X(0) = '1' and X(1) = '1' and EX_1_STATE = st0 then
               EX_0_NEXT_STATE <= st0;
            elsif X(3) = '1' and EX_1_STATE = st1 then
               EX_0_NEXT_STATE <= st0;
            elsif X(5) = '1' and EX_1_STATE = st2 then
               EX_0_NEXT_STATE <= st0;
            elsif X(3) = '0' and X(4) = '1' and EX_1_STATE = st1 then
               EX_0_NEXT_STATE <= st1;
            elsif X(5) = '0' and EX_1_STATE = st2 then
               EX_0_NEXT_STATE <= st1;
            elsif X(0) = '1' and X(2) = '1' and EX_1_STATE = st3 then
               EX_0_NEXT_STATE <= st1;
            elsif X(2) = '0' and EX_1_STATE = st3 then
               EX_0_NEXT_STATE <= st1;
            elsif X(0) = '0' and X(2) = '1' and EX_1_STATE = st3 then
               EX_0_NEXT_STATE <= st1;
            elsif X(3) = '0' and X(4) = '0' and EX_1_STATE = st1 then
               EX_0_NEXT_STATE <= st2;
            end if;
         when st1 =>
            if X(5) = '1' and EX_1_STATE = st2 then
               EX_0_NEXT_STATE <= st0;
            elsif X(4) = '1' and EX_1_STATE = st1 then
               EX_0_NEXT_STATE <= st0;
            elsif X(1) = '0' and EX_1_STATE = st0 then
               EX_0_NEXT_STATE <= st1;
            elsif X(1) = '1' and EX_1_STATE = st0 then
               EX_0_NEXT_STATE <= st1;
            elsif X(5) = '0' and EX_1_STATE = st2 then
               EX_0_NEXT_STATE <= st1;
            elsif X(4) = '0' and EX_1_STATE = st1 then
               EX_0_NEXT_STATE <= st1;
            elsif X(1) = '0' and EX_1_STATE = st3 then
               EX_0_NEXT_STATE <= st1;
            elsif X(1) = '1' and EX_1_STATE = st3 then
               EX_0_NEXT_STATE <= st1;
            end if;
         when st2 =>
            if X(2) = '1' and EX_1_STATE = st0 then
               EX_0_NEXT_STATE <= st0;
            elsif X(2) = '0' and EX_1_STATE = st0 then
               EX_0_NEXT_STATE <= st0;
            end if;
```

```
      end case;
   end process;

   -- Process to hold combinational logic of EX_1
   EX_1_COMBIN: process (EX_1_STATE, EX_0_STATE, X)
   begin
      case EX_1_STATE is
         when st0 =>
            if X(0) = '1' and X(1) = '0' and EX_0_STATE = st0 then
               EX_1_NEXT_STATE <= st0;
            elsif X(0) = '0' and X(1) = '0' and EX_0_STATE = st0 then
               EX_1_NEXT_STATE <= st1;
            elsif X(1) = '1' and EX_0_STATE = st1 then
               EX_1_NEXT_STATE <= st1;
            elsif X(2) = '1' and EX_0_STATE = st2 then
               EX_1_NEXT_STATE <= st1;
            elsif X(0) = '0' and X(1) = '1' and EX_0_STATE = st0 then
               EX_1_NEXT_STATE <= st2;
            elsif X(1) = '0' and EX_0_STATE = st1 then
               EX_1_NEXT_STATE <= st2;
            elsif X(2) = '0' and EX_0_STATE = st2 then
               EX_1_NEXT_STATE <= st2;
            elsif X(0) = '1' and X(1) = '1' and EX_0_STATE = st0 then
               EX_1_NEXT_STATE <= st3;
            end if;
         when st1 =>
            if X(3) = '1' and EX_0_STATE = st0 then
               EX_1_NEXT_STATE <= st0;
            elsif X(3) = '0' and X(4) = '1' and EX_0_STATE = st0 then
               EX_1_NEXT_STATE <= st0;
            elsif X(3) = '0' and X(4) = '0' and EX_0_STATE = st0 then
               EX_1_NEXT_STATE <= st0;
            elsif X(4) = '1' and EX_0_STATE = st1 then
               EX_1_NEXT_STATE <= st1;
            elsif X(4) = '0' and EX_0_STATE = st1 then
               EX_1_NEXT_STATE <= st1;
            end if;
         when st2 =>
            if X(5) = '1' and EX_0_STATE = st1 then
               EX_1_NEXT_STATE <= st0;
            elsif X(5) = '0' and EX_0_STATE = st1 then
               EX_1_NEXT_STATE <= st0;
            elsif X(5) = '1' and EX_0_STATE = st0 then
               EX_1_NEXT_STATE <= st2;
            elsif X(5) = '0' and EX_0_STATE = st0 then
               EX_1_NEXT_STATE <= st2;
            end if;
         when st3 =>
            if X(0) = '1' and X(2) = '1' and EX_0_STATE = st0 then
               EX_1_NEXT_STATE <= st0;
            elsif X(1) = '0' and EX_0_STATE = st1 then
               EX_1_NEXT_STATE <= st0;
            elsif X(0) = '0' and X(2) = '1' and EX_0_STATE = st0 then
               EX_1_NEXT_STATE <= st1;
            elsif X(2) = '0' and EX_0_STATE = st0 then
               EX_1_NEXT_STATE <= st2;
            elsif X(1) = '1' and EX_0_STATE = st1 then
               EX_1_NEXT_STATE <= st3;
            end if;
      end case;
   end process;
```

```
   -- Process to hold memory elements (flip-flops)
   MEMORY: process (CLK)
   begin
      if CLK'event and CLK='1' then
         EX_0_STATE <= EX_0_NEXT_STATE;
         EX_1_STATE <= EX_1_NEXT_STATE;
      end if;
   end process;
end BEHAVIOR;
```

Appendix C Software Bundle on Included CD

The attached CD contains decomposition software that was developed as a part of work on this thesis. The CD includes ready-to-run binaries of both web-based GUI and command-line versions of the software, source files and API documentation. The CD has the following structure:

/applet	GUI version of the software (Java applet)
/cmd	Command-line version of tools
/data	Sample input data files and the benchmark set [9]
/docs	API documentation
/src	Source files of the software
master.pdf	This Master's Thesis