INTL JOURNAL OF ELECTRONICS AND TELECOMMUNICATIONS, 2010, VOL. 56, NO. 1, PP. 13-24
Manuscript received February 02, 2010; revised March, 2010. 10.2478/v10177-010-0002-7

Statechart-based Controllers Synthesis in FPGA Structures with Embedded Array Blocks

Grzegorz Łabiak, Grzegorz Borowik

***Abstract*—Statechart diagrams, in general, are visual formalism for description of complex systems behaiour. Digital controllers, which act as reactive systems, can be very conveniently modeled with statecharts and efficiently synthesized in modern programmable devices. The paper presents in details syntax and semantics of statecharts and new implementation scheme. The issue of statecharts synthesis is not still ultimately solved. Main feature of the presented approach is the transformation of statechart diagrams into Finite State Machine, and through KISS format, functional decomposition and mapping into Embedded Memory Blocks. Embedded Memory are part of the modern programmable devices.**

***Keywords*—digital controller, statechart, FSM, decomposition, FPGA, symbolic methods, embedded memory.**

I. Introduction

DIGITAL controller design is a process which begins with informal description behavior and finishes with implementation in electronic devices [1]. First step of the process is to transform informal specification into formal one. This step is the most difficult, hence many formalisms have emerged, from very simple to very sophisticated ones, where statechart diagrams seem to be most efficient. Implementation process in modern programmable devices, especially equipped with Embedded Arrays Block or Configure Logic Blocks, requires using dedicated new methods [2], [3]. The traditional ones, like Espresso, are inefficient, sometimes giving results worse than without minimization. Presented in the paper new algorithm uses functional decomposition.

Digital controller acts like a reactive system. Such controller can be designed as traditional FSM, but this approach exhibits state explosion problem. Mainly, in case of modeled behavior, which features concurrency, number of states of the FSM grows exponentially. To cope with this inconvenience designer can use Petri net model, which directly allows to describe concurrency. State explosion is not the only problem in modeling complex behavior. Abundance of states, events, transition and parallel dependencies, makes that diagram become unclear. Then, good engineer practice is to divide modeled complex behavior into simpler sub-behaviors, according to classic paradigm: divide and conquer. Proceeding in this way, designer treats the modeled behavior like a tree of hierarchically connected sub-behaviors. This approach is supported by statechart diagrams.

The statechart diagrams were developed as a visual formalism for complex systems [4]. It is a state-based graphical notation which can be perceived as an extension of state transition graph of traditional finite state machine. In comparison to FSM they are enhanced with concurrency, hierarchy and broadcasting mechanism. At present time statecharts, also called state machines, are mainly used in UML technology [5], where are employed in behavior modeling of program objects (in sense of C++ or Java language).

The issue of hardware synthesis of statecharts is not solved ultimately. There are many implementation schemes, depending on target technology. Historically, first methodology published by [6], consists in transformation of statechart into set of hierarchically linked FSMs. Next, these FSMs can be implemented traditionally. Other approach is presented by [7] and is targeted at PLA structures, where main idea is to code craftily statechart configurations. The drawback of this method is that diagram can model only transitions between simple states. In 1999 [8] enhanced Drusinsky coding scheme by introducing so called prefix-encoding. Common drawbacks of the presented methods are lack of support for history attribute and broadcast mechanism. It is worth to mention other implementation methods such as using HDLs [9] or presented by [10] which is based on ASIP (Application Specific Instruction Processor).

The proposed Authors' design method, developed partly in HiCoS system [11], directly transforms behaviour specified with statecharts into Register Transfer Level, where register file codes global state. The transformation is realized as one-hot mapping [12], this means that one state corresponds to one flip-flop. Next, by means of symbolic methods [13], the RTL-like description is transformed into FSM form, which can be implemented in embedded array blocks in modern programmable devices [14], [15].

The rest of the paper is organized as follows. Section 2 presents syntax of statecharts, where main feature is modularity (ie the transitions cannot cross states borders). Section 3 describes an industry plant which is controlled by statechart-based controller. This more complicated example, where some syntax and semantics issues are presented in detail, is at the same time a bench mark. Section 4 presents semantics of dynamic behavior, mainly response of statechart-based digital circuit to an external and internal events. Section 5 describes digital synthesis of statechart-based controllers realized by means of flip-flops and Boolean functions. Section 6 presents

This work has been supported by the European Union in the framework of European Social Fund through the Warsaw University of Technology Development Programme and by Ministry of Science and Higher Education financial grant no. N517 003 32/0583.

G. Łabiak is with the Computer Eng. & Electronics Dept., University of Zielona Góra, Podgórna 50, 65-246 Zielona Góra, Poland (e-mail: G.Labiak@iie.uz.zgora.pl).

G. Borowik is with the Institute of Telecommunications, Warsaw University of Technology, Nowowiejska 15/19, 00-665 Warsaw, Poland (e-mail: G.Borowik@tele.pw.edu.pl).

the transformation of statecharts into FSM described in KISS format. Section 7 describes the idea of ROM-based synthesis algorithm and mapping into Embedded Memory blocks, and section 8 presents experiments results.

II. Syntax of Statecharts

The statechart diagrams [4] have been devised in order to improve the specification of reactive systems of complex behavior. It is a state-based graphical notation which enhances the traditional finite automata with concurrency, hierarchy and broadcast mechanism. States are connected by arcs with predicates. A complex state can be assigned a group of states (simply or complex), thus creating hierarchy relationships. States can be in a concurrency relationship. An activity can be removed from subordinated states in the exception style through firing transition from their ancestor. The presence of the final state (in the diagram bull's eye) prevents exception transitions, unless the final state is active.

The big problem with statecharts is syntax and semantics. A variety of application domains caused that many authors proposed their own syntax and semantics [16], sometimes differing significantly. Syntax and semantics presented in this paper are intended for specifying the behaviour of binary digital controllers which would satisfy as much as possible the UML statndard. The selection of language characteristics was made based on application domain and the technological constrains of programmable logic devices.

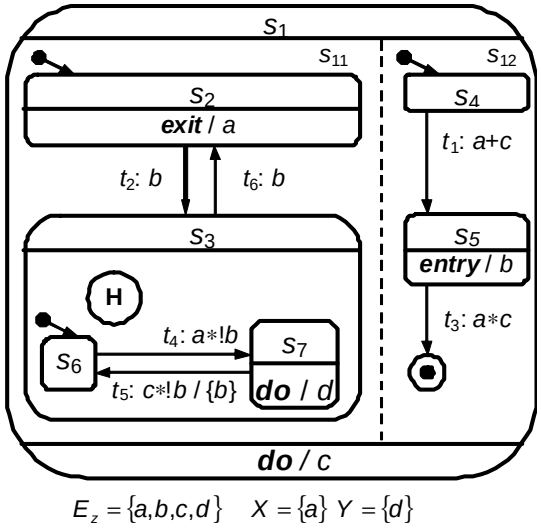

Fig. 1. Example of Statechart diagram.

As a result of those considerations it was assumed that syntax of statecharts HiCoS is to be intended for untimed control systems which operate on binary values. Hence, our statecharts feature hierarchy and concurrency, simple state, composite state, end state, discrete events, actions assigned to state (*entry*, *do*, *exit*), simple transitions, history attribute and logic predicates imposed on transitions, whereas cross-border transitions are forbidden. Another very essential issue is to allow the use of feedbacks, it means that events generated in a circuit can affect its behavior. The role of an end state is to prevent removing away an activity from a sequential automaton before the end state became active. Such elements as factored transition paths and time were rejected. Other syntax characteristic like cross-level and composite transitions, synch states have been shifted to the farther stage of the research. An example of statechart is depicted in figure 1, where event $a \in X$ is an input to the system and event $b \in Y$ is an output. Events *b* and *c* are of local scope.

III. Chemical Reactor – Case Study

The industrial reactor, for the first time presented in [17] is a part of hydraulic-mechanical plant, whose functioning is governed by a discrete controller (Fig. 2). The reactor measures out two substances, mixes them together and pours the product into the wagon which transports the outcome to its destination station.

A. Reactor Working Description

The detailed working of the reactor is as follows. Initially the reacting substances are kept in containers SV1 and SV2. The emptied wagon waits in its initial position on the right. After the signal *x0* a technological cycle starts: valves *y1* and *y2* are opened and scales MV1 and MV2 are poured in, and wagon starts moving to the left (signal *y9*). The pouring of the substrates lasts until sensors *x1* and *x3* in the scales indicate exceeding upper limits. After both sensors indicate exceeding upper limits valves *y3* and *y4* emptying scales MV1 and MV2 become on simultaneously and the main reaction process starts. The agitator A is ready to switch on. After the substance in reactor main container R is over the sensor *x5*, the agitator becomes on. When the substance is beneath again, the agitator becomes off and ready to start again. While emptying scales and pouring main container R, the wagon is moving to its position on the left. Next, when scales MV1 and MV2 are emptied (which is indicated by sensors *x2* and *x4*) and in the meantime the wagon has reached its left position (sensor *x7*), the main container valve *y5* is opened and the container is emptied till the level of substance drops below the reading of the sensor *x6*. Next the wagon starts moving right (signal *y8*). When the wagon reaches the far right position (sensor *x9*), the wagon is emptied (valve *y6*). The end of emptying is signaled by *x9* sensor, and after that the technological cycle is finished and the whole plant is ready to start again.

B. Statechart Diagram

Fully modular statechart diagram is a diagram whose compound component behaviors are only defined by its sub-components. This generally means, that the transitions crossing the state borders are forbidden and broadcast events are also forbidden. The benefits of full modularity are clarity of the diagram and a more efficient verification. This stems from the fact that the dynamic properties of the complex behavior can be obtained from already verified properties of its component sub-behaviors.

Figure 3 presents statechart diagram of the controller. In this case it is good to model some of its component sub-behaviors,

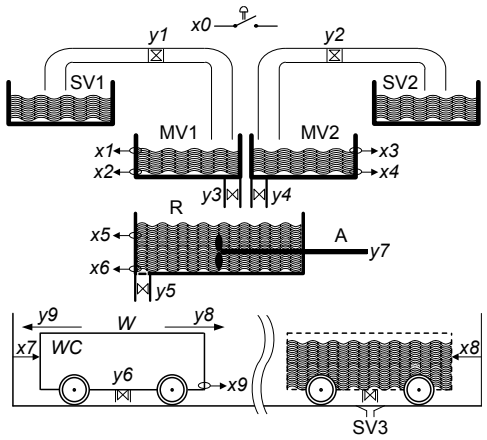

Fig. 2. Chemical reactor with wagon – Schematic diagram

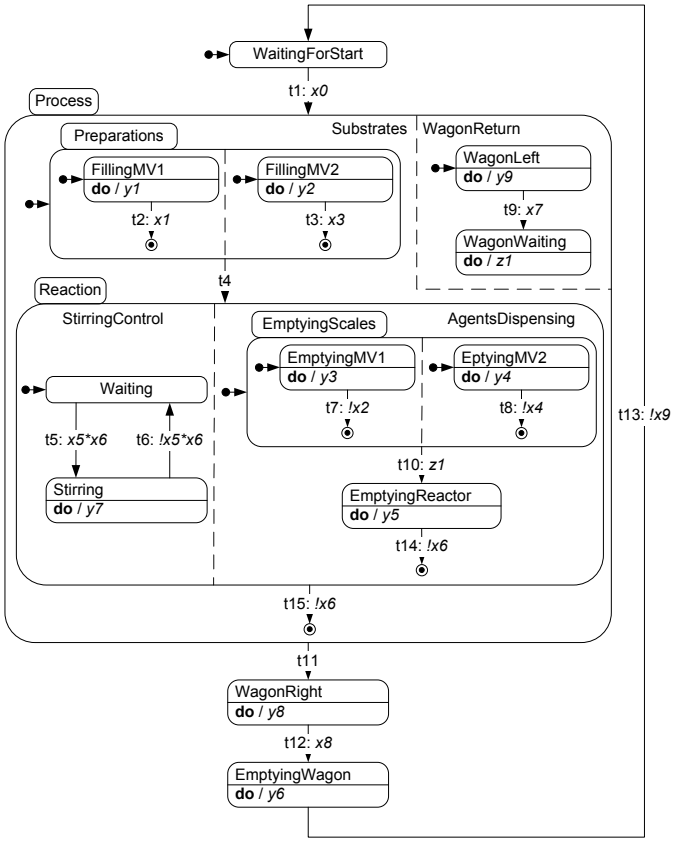

Fig. 3. Chemical reactor – statechart diagram.

namely the agitator control (simple states *Waiting* and *Stirring*), agents dispensing to main reactor container (consisting of one state *EmptyingReactor* and two completely parallel processes: states *EmptyinMV1*, *EndState* and *EmptyingMV2*, *EndState*) and wagon motion to the left (states *WagonLeft* and *WagonWaiting*). These three concurrent processes neither start nor finish at the same time. They are started with transition *t1* and *t4* and finished with *t10* and *t11*. This means that they overlap and must be synchronized with these four transitions. This is realized by means of variable *z1*, which is being broadcast in state *WagonWaiting* and is a predicate on transition *t10*. This behavior cannot be modeled fully modularly, internal variable *z1* must synchronize the three processes and the presence of the variable slightly disturbs modularity.

IV. Semantics

A digital controller specified with a Statechart and realized as an electronic circuit is meant to work in an environment which prompts the controller by means of events. It is assumed that every event (incoming, outgoing and internal) is bound with a discrete time domain. The controller is reacting to the set of accessible events in the system through firing a set of enabled transitions called a microstep. Because of feedback, execution of a microstep entails generating farther events and causes firing subsequent microsteps [18]. Events triggered during a current microstep do not influence transitions being realized, but are only allowed to affect behavior of a controller in the next tick of discrete time, that is, in the next microstep. A sequence of subsequently generated microsteps is called a step and additionally it is assumed that during a step no events can come from the outside world. A step is said to be finished when there are no enabled transitions. Figure 4 depicts a step which consists of two simple microsteps. After the step is finished the system is in state *STOP*. Summarizing, dynamic characteristics of hardware implementation are as follows:

- system is synchronous,
- system reacts to the set of available events through transition executions,
- generated events are accessible to the system during next tick of the clock.

In figure 4 a simple diagram and its waveforms illustrate the assumed dynamics features. When transition t_1 is fired (T = 350) event t_1 is broadcasted and becomes available to the system at next instant of discrete time (T = 450). The activity moves from state *START* to state *ACTION*. Now transition t_2 becomes enabled. Its source state is active and predicate imposed on it (event t_1) is met. So, at instant of time T = 450 the system transforms activity to the state *STOP* and triggers event t_2, which does not affect any other transition. The step is finished.

V. Statechart Diagrams Synthesis

Statechart diagram synthesis is process by which controller described by statecharts is turned into design implementation in terms of logic gates and flip-flops.

A. Foundations of Hardware Implementation

The main assumption of a hardware implementation behaviour described with statecharets diagram is that the systems specified in this way can directly be mapped into programmable logic devices. This means that elements from a diagram (for example states or events) are to be in direct correspondence with resources available in a programmable device — mainly flip-flops and programmable combinatorial logic. Basing on that assumption and taking into account assumed dynamic characteristics, following foundations of hardware implementation has been formulated:

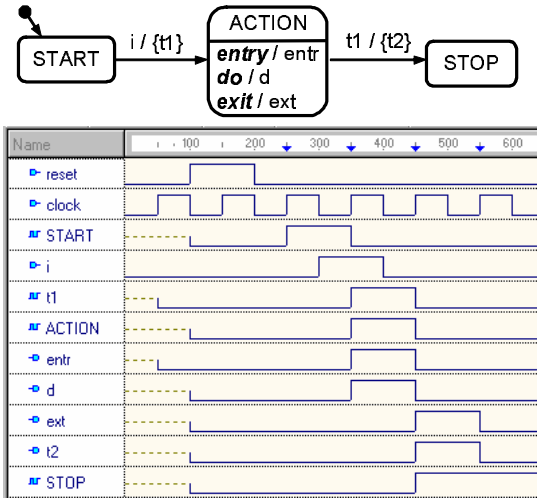

Fig. 4. Simple diagram and its waveform.

- each state is assigned one flip-flop — activity means that state associated with the flip-flop can be active or in case of a state with history attribute is remembered its past activity; activity of state is established on the basis of activity of flip-flops assigned to superordinate states (in sense of hierarchy),
- each event is also assigned one flip-flop — activity means occurrence of associated event and it is sustained to the next tick of discrete time when the event becomes available do the system,
- based on diagram topography and rules of transition executions, excitation functions are created for each flip-flop in a circuit.

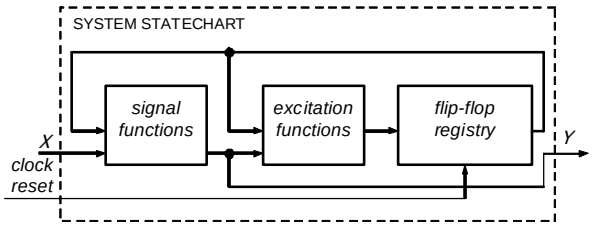

Fig. 5. Statechart diagrams Hardware Implementation

Farther statechart diagrams synthesis description is mainly revolving around specification of flip-flop excitation functions of two types: state flip-flops and event flip-flops.

B. State Flip-flop Excitation Functions

Every state is assigned one flip-flop. Logic **1** on its output means occurrence of one of two situations:

- activity of state (to whom the flip-flop was assigned),
- remembering that the state was most recently active — this takes place in case of states with history attribute.

These two circumstances are essentially different. Therefore it is necessary to define the rules which will allow to determine the former and the latter situation in an unambiguous way. As far as activity of a state is concerned, this is realized on the basis of activity of flip-flops assigned to the superordinate states. The state is said to be active when every flip-flop bound with the states belonging to the path (in sense of hierarchy tree) carried from the state to the root state (located on top of a hierarchy) is asserted. Formally activating condition is calculated in the following way:

$$activecond(s) = \prod_{s_i \in path(root,s)} s_i \qquad (1)$$

where s_i is a signal from flip-flop output.

Having established a role that flip-flop is to play in a digital circuit it is possible to formulate general assumption regarding its excitation function. This function yields **1** when the state bound with given flip-flop is:

- not active and in next iteration will be active,
- an active state or is, so called, a recently active state (it can take place in case of state with history attribute) and in next iteration will also be active or recently active.

One characteristic feature of these two assumptions is causal relationship which consist in that before state became most recently active it must be prior active. This observation leads to state flip-flop excitation function of following shape:

$$\delta(s) = \underbrace{activate(s)}_{\text{a}} + \underbrace{s * \overline{inactivate(s)}}_{\text{b}} \qquad (2)$$

where *a* and *b*, respectively, are:

a) activating component (*activate*): it assumes logic **1** when the state bound with a flip-flop is not active and in next iteration will become active; this corresponds to the situation when directly incoming transitions to the state fires or, in case of default states, the state activated is by directly superordinate state,

b) sustaining component: it assumes logic **1** when 1) the state bound with a flip-flop is active (it means that component *activate* must have been fulfilled before) and in next iteration the state will also be active or 2) in case of the state is attributed history property the state is recently active and in next iteration will also be recently active; factor *inactivate* assume value **1**, when the state loses activity and at the same time is not recently active, this corresponds to the situation when one of some output transitions is fired.

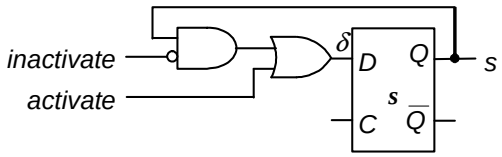

Fig. 6. Logic diagram of flip-flop excitation function.

A variable s in equation 2, is a feedback signal and its role is to sustain flip-flop activity since the moment specified by *activate* component till the moment determined by *inactivate* factor. The excitation function defined in that way leads to the logic diagram presented in figure 6. Farther descriptions of synthesis rules of state flip-flop excitation function are focused around detailed definition of activating components and inactivating factor.

1) Activate Component: The conclusion which results from what is stated in point a) is that to form an expression for component *activate* is necessary to investigate two cases.

- The first case holds when a normal state (i.e. not being a default state) is examined. Then activation of such state depends only on transitions which directly come to the state. Formally, it is defined by following equation:

$$activate(s) = \sum_{t_i \in {}^\bullet s} encond(t_i) \tag{3}$$

- Second case takes place when it comes to a state which is the default state. Then activation of such a state depends not only on directly incoming transition, but additionally depends on activity of directly superordinate state. When directly superordinate state becomes active it means that one of its directly subordinate states must also become active. It is as if activity "comes from above". Equation which describes this case presents as follows:

$$\begin{aligned} activate(s_d) &= \underbrace{\sum_{t_i \in {}^\bullet s} encond(t_i)}_{\text{a}} + \\ &+ \underbrace{\prod_{s_j \in path(root, parent(s_d))} \delta(s_i)}_{\text{b}} * \\ &* \underbrace{\overline{\sum_{s_i \in hrc(parent(s_d))} s_i}}_{\text{c}} \end{aligned} \tag{4}$$

Component *a* is responsible for activating which comes from directly incoming transition, similarly as it is in previous case (eq. no. 3). Factor *b* is referring to the activation which is caused by directly superordinate state. This factor is, in fact, modified activating condition (see eq. no. 1). δ is a flip-flop activating function and as a signal is taken from flip-flop input. This means that factor *b* represents activity of the parent state in next tick of discrete time. Factor *c* allows to activate flip-flop only when there is no other active flip-flop. Activity of other flip-flop means that default state flip-flop must not be activated, because the other flip-flop remembers past activity of the other state to which is bound with.

2) Inactivate Factor: This factor is to assume value **1** when the state to which a flip-flop is bound with is to lose activity and at the same time is not recently active state. The situation arise in consequence of firing some output transitions. The issue is which are these output transitions. This depends both on whether investigated state is an end state or a normal state (in this context normal state means not an end state) and on presence of history attribute and also on whether investigated state belongs to an automaton with an end state. The latter case means that transitions of higher levels of hierarchy cannot take activity. First, to put it in an order, let us consider four cases when state is a normal state and has or not history attribute.

- state without history attribute belonging to the automaton without an end state:

$$\begin{aligned} inactivate(s) &= \underbrace{\sum_{t_i \in s^\bullet} encond(t_i) +}_{\text{a}} \\ &= \underbrace{\overline{\prod_{s_i \in path(root, parent(s))} \delta(s_i)}}_{\text{b}} \end{aligned} \tag{5}$$

Flip-flop is reset by firing of directly outgoing transitions from state *s* (component *a*) or by firing output transitions of superordinate states. The latter case, in equation, is represented by negated activity condition of directly superordinate state (component *b*).
- state without history attribute belonging to an automaton with an end state,
- state with history attribute belonging to automaton without an end state,
- state with history attribute belonging to automaton with an end state:

$$inactivate(s) = \sum_{t_i \in s^\bullet} encond(t_i) \tag{6}$$

In these three cases only directly outgoing transitions can reset a flip-flop. This results from semantics and from taken assumption as to what role plays activity of a flip-flop.

Now there is an end state case left to investigate. An end state can have or not history attribute, so it gives two next situations to analyze:

- end state without history

$$inactivate(s) = \underbrace{\overline{\prod_{s_i \in path(root, parent(s))} \delta(s_i)}}_{\text{b}} \tag{7}$$

An end state must not have directly outgoing transitions, but its activity can be taken by transitions of higher levels of hierarchy (see also eq. no. 5).
- end state with history attribute:

$$inactivate(endst) = 0 \tag{8}$$

Attribution of history property to an end state means that a flip-flop bound with such a state will never be reset, because an end state must not have directly outgoing transitions and reactivating automaton with such a state, as result of firing transition of higher levels of hierarchy, will always cause activating the end state. In general, ascribing history attribute to the automaton with an end state makes no sense and hence is not recommended.

C. Event Flip-flop Excitation Functions

Hardware implementation in *FPGA* structures is based on the assumption that the circuit responds to the set of currently available events in next tick of discrete time. Between an event and a respond to it there is a period of time equal to a period of clock signal. To fulfill this assumption there is a necessity to bound with every place in circuit where event

can be triggered one flip-flop. The flip-flop's assignment is to sustain information about an event for the clock signal period. Basically, there are four possible places where an event can be generated:

1) Transition: Firing transition can be assigned broadcasting set of events. Excitation function of such a flip-flop is a simple enabling transition condition:

$$\delta(e_t) = encond(t) \tag{9}$$

2) Entry Action: Every state can be assigned an entry action, which is executed when state is being activated. This is, of course, broadcast set of events. Activation of a state takes place when, at given moment of discrete time, state is not active (factor *a*) and at next instant of time will become active (factor *b*):

$$\delta(e_{en}) = \underbrace{\overline{activecond(s)}}_{\text{a}} * \underbrace{\prod_{s_i \in path(root,s)} \delta(s_i)}_{\text{b}} \tag{10}$$

3) Do Action: Sometimes called static action is a set of events which are broadcast at every tick of clock signal, as long as state to which the action is ascribed is active. Therefore, an excitation function boils down to the state flip-flop excitation function (see section V-B).

4) Exit Action: This action complements entry action and is being executed when given state is active (factor *a*) and at next instant of time will lose activity (factor *b*):

$$\delta(e_{ex}) = \underbrace{activecond(s)}_{\text{a}} * \underbrace{\overline{\prod_{s_i \in path(root,s)} \delta(s_i)}}_{\text{b}} \tag{11}$$

VI. Statechart Diagrams Transformation into FSM Model

Transformation of statechart diagrams into FSM model consists in constructing equivalent finite state machine, which for external observer behaves just the way statechart does.

Definition 1: Finite state machine of Mealy type, denoted as M, is a following tuple:

$$M = \langle X, S, Y, \delta_M, \lambda_M \rangle$$

where:

$$\begin{aligned} X &= \{x_1, \ldots x_m\} - \text{set of input signals} \\ S &= \{s_1, \ldots s_n\} - \text{set of internal states} \\ Y &= \{y_1, \ldots y_r\} - \text{set of output signals} \end{aligned}$$

and:

$$\begin{aligned} \delta_M : D_{\delta_M} &\mapsto S - \text{is a transition function} \\ \lambda_M : D_{\lambda_M} &\mapsto Y - \text{is a output function} \\ D_{\delta_M} &\subseteq X \times S \\ D_{\lambda_M} &\subseteq X \times S \end{aligned}$$

When $D_{\lambda_M} \subseteq S$ the automaton is of Moore type.

The construction involves building equivalent Moore-type automaton from statechart elements, where members of the sets are explicitly enumerated and functions are given symbolically in tabular form, eg KISS format [19].

A. Statechart as a FSM

Statechart diagrams in some sense can be perceived as a finite state machine of Moore type. Then, such a FSM formally can be defined as follows:

Definition 2: A statechart FSM (SFSM) is defined as a 6–tuple: $SFSM = (m, n, r, \delta_S, \lambda_S, Init)$ where: m is the number of Boolean inputs, n is the number of Boolean state variables (all flip-flops), r is the number of Boolean outputs, δ_S is the functional vector of the state transition functions $\delta_{Si} : \{0,1\}^m \times \{0,1\}^n \rightarrow \{0,1\}$ and $1 \leq i \leq n$, λ_S is the functional vector of the output functions $\lambda_{Sj} : \{0,1\}^n \rightarrow \{0,1\}$ and $1 \leq j \leq r$, $Init$ is the initial state.

For the diagram from figure 1 $m = 1$, $n = 13$, $r = 1$. Because output function λ_S depends only on state variables, statechart FSM is of Moore type. The set of equivalent FSM input signal is the set X of statechart input events, the set of FSM output signals is the set Y of statechart event visible to the environment. Cardinality of both sets is, respectively, $|X| = m$ and $|Y| = r$. For example for the diagram from fig. 1 $X = \{a\}$ and $Y = \{d\}$. The set S of equivalent FSM states, transition function δ_M and output function λ_M need referring to statechart state transition graph (STG, see fig. 7). STG presents global states of statechart and transitions between them. The transition is fired when events imposed on it are being broadcast, whether coming from environment (bolded in the fig. 7) or generated internally in the diagram. Then, the set S of equivalent FSM states is the set of all statechart global states from STG (see def. 3), FSM transition function δ_M is a relationship between STG transition source and target states, FSM output function λ_M is a relationship between STG global states and events assigned to them which are visible to the environment (bolded in the fig. 7).

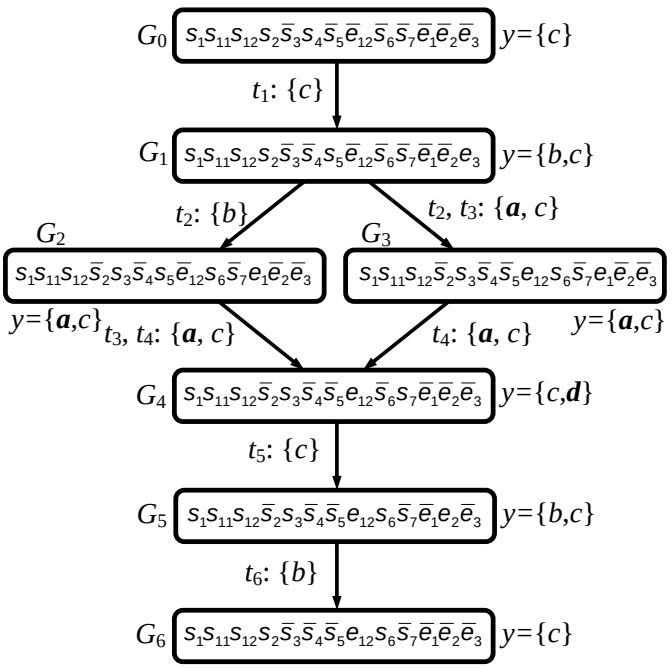

Fig. 7. Statechart state transition graph for the diagram from fig. 1.

B. Global State of a Statechart

The global state of the statechart is defined as follows:

Definition 3: Global state G is a set of all flip-flops' states in the system, bound both with local states and with distributed events, which can be generated in several parts of the diagram and separately memorized.

Figure 7 presents state transition graph for the diagram from figure 1. The diagram consists of 10 state flip-flops $\{s_1, s_{11}, s_{12}, s_2, s_3, s_4, s_5, e_{12}, s_6, s_7\}$ and 3 event flip-flops $\{e_1, e_2, e_3\}$. The flip-flop denoted as e_1 corresponds to the exit event a assigned to state s_2, e_2 corresponds to the entry action b assigned to state s_5, and e_3 corresponds to the transition action (broadcasting of event b) bound with transition t_5 firing (from state s_7 to state s_6). Global state comprises all information about the statechart both about currently active states and their past activity. Number of global states is 7. Technically, global state can be expressed as a conjunction formula, where variable in formula are bound with flip-flops' output signal: $G_0 = s_1\ s_{11}\ s_{12}\ s_2\ \overline{s_3}\ s_4\ \overline{s_5}\ \overline{e_{12}}\ \overline{s_6}\ \overline{s_7}\ \overline{e_1}\ \overline{e_2}\ \overline{e_3}$.

C. Characteristic Function

To implement the algorithm of transformation statechart FSM into FSM there are needed means to represent symbolically set of global states. The set of states can be implemented by means of notion of characteristic function represented by Boolean equation.

Definition 4: A *characteristic function* χ_A of a set of elements $A \subseteq U$ is a Boolean function $\chi_A : U \to \{0,1\}$ defined as follows:

$$\chi_A(x) = \begin{cases} 1 \Leftrightarrow x \in A \\ 0 \Leftrightarrow x \notin A. \end{cases} \tag{12}$$

The characteristic function is calculated as a disjunction of all elements of *A* (i.e. the set of all global states). Operations on sets are in direct correspondence with operations on their characteristic functions. Thus: $\chi_{(A \cup B)} = \chi_A + \chi_B$, $\chi_{(A \cap B)} = \chi_A * \chi_B$, $\chi_{(\overline{A})} = \overline{\chi_A}$, $\chi_{(\emptyset)} = 0$. The characteristic function allows sets to be efficiently represented in computer memory by means of BDDs [20]. For example, if, for the state transition graph from figure 7, the global state is defined as a conjunction of flip-flop variable (eg. initial state $G_0 = s_1\ s_{11}\ s_{12}\ s_2\ \overline{s_3}\ s_4\ \overline{s_5}\ \overline{e_{12}}\ \overline{s_6}\ \overline{s_7}\ \overline{e_1}\ \overline{e_2}\ \overline{e_3}$) then the characteristic function of the set of all global states is a disjunction of respective Boolean expressions, where one expression represents one global state:

$$\begin{aligned}
\chi_G &= G_0 + G_1 + G_2 + G_3 + G_4 + G_5 + G_6 = \\
&= s_1\ s_{11}\ s_{12}\ s_2\ \overline{s_3}\ s_4\ \overline{s_5}\ \overline{e_{12}}\ \overline{s_6}\ \overline{s_7}\ \overline{e_1}\ \overline{e_2}\ \overline{e_3} + \\
&= s_1\ s_{11}\ s_{12}\ s_2\ \overline{s_3}\ \overline{s_4}\ s_5\ \overline{e_{12}}\ \overline{s_6}\ \overline{s_7}\ \overline{e_1}\ \overline{e_2}\ e_3 + \\
&= s_1\ s_{11}\ s_{12}\ \overline{s_2}\ s_3\ \overline{s_4}\ s_5\ \overline{e_{12}}\ s_6\ \overline{s_7}\ e_1\ \overline{e_2}\ \overline{e_3} + \\
&= s_1\ s_{11}\ s_{12}\ \overline{s_2}\ s_3\ \overline{s_4}\ \overline{s_5}\ e_{12}\ s_6\ \overline{s_7}\ e_1\ \overline{e_2}\ \overline{e_3} + \\
&= s_1\ s_{11}\ s_{12}\ \overline{s_2}\ s_3\ \overline{s_4}\ \overline{s_5}\ e_{12}\ \overline{s_6}\ s_7\ \overline{e_1}\ \overline{e_2}\ \overline{e_3} + \\
&= s_1\ s_{11}\ s_{12}\ \overline{s_2}\ s_3\ \overline{s_4}\ \overline{s_5}\ e_{12}\ s_6\ \overline{s_7}\ \overline{e_1}\ e_2\ \overline{e_3} + \\
&= s_1\ s_{11}\ s_{12}\ s_2\ \overline{s_3}\ \overline{s_4}\ \overline{s_5}\ e_{12}\ s_6\ \overline{s_7}\ \overline{e_1}\ \overline{e_2}\ \overline{e_3}
\end{aligned} \tag{13}$$

D. Transition Relation

Transition relation is a relation which relates transition's source and target states with an event (or events) imposed on this transition. This notion symbolically can be expressed by means of characteristic function:

Definition 5: A characteristic function χ_{δ_S} of the transition relation of the functional vector δ_S is a Boolean function $\chi_{\delta_S} : \{0,1\}^n \times \{0,1\}^m \times \{0,1\}^n \to \{0,1\}$ defined as follows:

$$\chi_{\delta_S}(s', x, s) = \begin{cases} 1 & \Longleftrightarrow\ s' = \delta_S(x, s)\ , \\ 0 & \text{otherwise.} \end{cases}$$

where s', s and x are sets of state variables of next state, present state, and input signal variables, respectively.

In practice, the function χ_{δ_S} is calculated using the following equation:

$$\chi_{\delta_S}(s', x, s) = \prod_{i=1}^{n} [s_i' \odot \delta_{Si}(x, s)] \tag{14}$$

where the symbol $\odot$ represents the logic XNOR operator and n is the number of state variables.

The relation $\chi_{\delta_S}(s', x, s) = 1$ implies that in state s there exists a transition to state s' on input x.

E. Symbolic State Space Generation

Having defined notions of statechart global state and characteristic function it is necessary to present an algorithm which computes symbolic state space of statechart, where set of global states symbolically is represented by its characteristic function.

The algorithm traverses the state transition graph of statechart FSM in breadth-first manner, moving from a set of current states to the set of its fan-out states. In this approach a sets of states are represented by means of characteristic functions. The key operation required for traversal is the computation of the range of a function, given a subset of its domain. The symbolic state space exploration of statecharts relies in [21]:

- association excitation functions to state and event flip-flops,
- association logic functions to signals,
- representation of Boolean function as BDDs,
- representation of sets of states using their characteristic functions,
- computation of a set of next states as an image of the state transition function on the current states set for all input signals.

Starting from the default global state and the set of signals, symbolic state exploration methods enable the computation of the entire set of next global states in one formal step. The symbolic state space algorithm of statechart is presented below.

Listing 1. Symbolic traversal of state space

```
1 symb_trav_of_Statechart(Z, init_mark) {
    χ[G0⟩ = curr_mark = init_mark;
3   while (curr_mark != 0) {
      next_mark = im_comp(Z, curr_mark);
```

```
5       curr_mark = next_mark* χ[G0⟩ ;
        χ[G0⟩ = curr_mark + χ[G0⟩ ;
7     }
  }
```

The variables in italics represent characteristic functions of corresponding sets of configurations. All logical variables are represented by BDDs. Several subsequent global states are simultaneously calculated using the characteristic function of current global states and transition functions. This computation is realized by the *im_comp* function, which calculates image of function for given subset of the domain. The set of subsequent global states is calculated from the following equations:

$$next\_mark = \exists_s \exists_x (curr\_mark * \prod_{i=1}^{n} [s'_i \odot (curr\_mark * \delta_{Si}(s,x))]) \quad (15)$$

$$next\_mark = next\_mark\langle s' \leftarrow s \rangle \quad (16)$$

where s, s', x denote the present state, next state and input signals respectively; $\exists_s$ and $\exists_x$ represent the existential quantification of the present state and signal variables; n is a number of state variables; $\odot$ and $*$ represent logic operators XNOR and AND, respectively; equation 16 means swapping variables in expression.

For example a characteristic function of the set of all global states for the diagram from figure 1 is presented by equation 13.

F. The Transformation

The construction of a FSM state transition table of the function δ_M can be carried out in many ways. Authors' proposal is not an optimal approach, but is relatively simple in coding on the one hand and not so computationally complex on the other hand, that testing benchmarks has been conducted quickly and successfully. Construction of FSM state transition table consists in checking for every pair of statechart global states whether exists transition between them (see listing 2), and if exists, in calculating an event or a set of events imposed on this transition. Result of the transformation, state transition table with outputs, is in KISS format [19].

1) FSM Textual Form: FSM-description is a textual form of Finite State Machine, also known as KISS format [19], which in tabular way defines FSM according to the following grammar:

```
.i <num-inputs>
.o <num-outputs>
.p <num-terms>
.s <num-states>
.r <reset-state>
<in> <current-st> <next-st> <out>
    . . .
```

where *num-inputs* is the number of inputs to the FSM, *num-outputs* is the number of outputs of the FSM, *num-terms* is the number of "<in> <current-st> <next-st> <out>" 4-tuples, *num-states* is the number of distinct states that appear in "<current-st>" and "<next-st>" columns, *reset-state* is the symbolic name of the start state. *in* is a sequence of *num-inputs* of $\{0, 1, -\}$, *out* is a sequence of *num-outputs* of $\{0, 1, -\}$. *current-st* and *next-st* are symbolic names for the current state and next state transitions of the FSM. Format of this type describes automaton as an automaton of Mealy type.

2) The Algorithm: The algorithm of transformation consists in establishing a map between statechart FSM and FSM-description. The FSM set of inputs (X) is equivalent to the sets of inputs in statechart model, the FSM set of outputs (Y) is equivalent the sets of outputs in statechart model. The elements of the set of states S in FSM are coded as a global state G in statechart (see definition 3). The parameters *num-inputs*, *num-outputs* from KISS format is a cardinality of the sets, respectively, X and Y. In case of transitions, from SFSM side we have automaton of Moore type and from FSM-description we have model of Mealy automaton, where transitions are explicitly enumerated. Hence, the transformation involves two major steps:

a) transformation Moore automaton into Mealy automaton,
b) generation, transition by transition, KISS 4-tuple.

The transformation Moore automaton into Mealy automaton is very simple [22]. If we omit the answer of Moore's automaton on empty input ε, this means in fact that we exclude answer of automaton at the first state, the two automata differ only in their output functions. Let λ be Moore automaton output function and λ' be Mealy automaton output function. Then we have:

$$\lambda'_M(x,s) = \lambda_M(\delta_M(x,s)) \quad (17)$$

and we must remember to add one extra state (the initial state) to the set of state of Mealy automaton and one extra transition from this extra state to the so far first state. Hence, the parameter *num-states* in KISS format is cardinality of set S plus 1. The transition is unconditional (empty input ε), and the output is an output from first state of Moore automaton.

As far as transition function δ_M is concerned, in format KISS, the function is given in tabular form; one 4-tuple for one transition. The problem is how to generate the elements of the 4-tuple.

The algorithm depicted in listing 2 starts with characteristic function of global states space $\chi_{[G_0\rangle}$ and with characteristic function χ_{δ_M} of functional vector δ_M. The transition, regarding input signals, is represented by product t, which is a relation between current (G_i) and next (G'_j) state, represented as conjunction formulae (line no. 4). For every pair of states: current state and next state (G_i, G'_j) is being checked whether there is a transition between them (line no. 5). In line no. 6 states variables (s, s') are removed from transition product t, hence t_x represents this part of the expression which solely depends on input variables x. Then current and next states are put into 4-tuple (lines 7 and 8). Between lines 11 and 17 is being computed input vector. For each minterm in t_x expression is being checked how this minterm depends on input variables. This is realized by means of logic differential, which can formally be computed according to the following

Listing 2. Transitions generation algorithm

```
Transitions_Generation(Z, χ[G0⟩ ,χδM ) {
  for each global state Gi χ[G0⟩ (Gi) = 1; {
    for each global state G'j χ[G0⟩ (G'j) = 1; {
      t = χδM (s', x, s) * Gi * G'j;
      if t = 0 then continue;
      tx = ∃s' ∃s t
      current−st = Gi;
      next−st = G'j⟨s' ← s⟩;
      for each minterm mi in tx {
        for each input xj ∈ X {
          if ∂mi/∂xj ≠ 0 then { // deps on xj
            if mi * xj ≠ 0 then in[j] = 1
            else in[j] = 0;
          }
          else in[j] = −;
        }
      }
      for each output yi ∈ Y {
        if G'j⟨s' ← s⟩ * λMi ≠ 0 then out[i] = 1
        else out[i] = 0;
      }
    }
  }
}
```

formula:

$$\frac{\partial f}{\partial x_i} = f_{x_i} \oplus f_{\overline{x}_i} \tag{18}$$

where f_{x_i}, $f_{\overline{x}_i}$ are, respectively, positive and negative algebraic cofactor and symbol $\oplus$ represents XOR operator. In lines 12, 13 and 15 is determined how signal x_j affects the transition. In lines from 18 to 21 output vector is being computed. Although this is not presented in the algorithm, it is enough to execute these 4 lines one time per next state s'.

G. The Example of the Transformation

Fig. 1 presents statechart diagram which is, on one hand, complex enough to show nearly every syntax feature and, on the other hand, simple enough to give and discuss its state diagram (fig. 7). The diagram, although very similar to FSM state transition graph, exhibits statecharts perspective. States are coded as a statechart global states, according to HiCoS implemental scheme. Depicted events are both input, output and internal, but input and output events are bolded. Names in labels at the transitions, put before colon, are transitions names (ex. t_1), which correspond to the transitions from the diagram in figure 1. Below is a FSM-description, where abstract names of states (ex. G_0, G_1) were preserved and one extra state (start) was added.

```
.i 1              0 G1   G2 0
.o 1              1 G1   G3 0
.p 8              1 G2   G4 1
.s 8              1 G3   G4 1
.r start          - G4   G5 1
- start  G0 0     - G5   G6 0
- G0     G1 0
```

VII. ROM-BASED SYNTHESIS

The presented implementation scheme (developed under HiCoS project) allows the transformation of statechart diagrams into FSM-description. Using this scheme we can implement FSMs in field programmable gate array devices.

Although most of the methods gathered and discussed in [23] can be effectively used for synthesis of FSM implemented with gates and flip-flops, they are not efficient for today's programmable structures, particularly for FPGA devices with embedded memory blocks [23]. Such implementations would benefit from a structure with a separate memory block which is common in microprogrammable circuits. However, an advanced apparatus for design of address modifier is required to support the synthesis based directly on the FSM transition table.

A limited size of embedded memory blocks available in FPGA devices is the main argument behind the application of this structure. For example, Altera FLEX family devices have 2048-bit EAB memory blocks. In [24] it is demonstrated that the ROM-based implementation of an example sequential circuit – the tbk benchmark – requires 16,384 bits of memory; this considerably exceeds the resources available in the FLEX 10K device. An alternative implementation of this circuit with LUTs requires 895 logic cells (a result from the Altera Quartus II ver. 8.1 system); this also exceeds the resources available in the FLEX 10K device, as it has only 576 cells. Thus, the tbk implementation with this device must rely on the new FSM architecture.

Clearly, a considerably larger number and size of embedded memory blocks in the newer programmable Stratix and Cyclone devices do not eliminate this problem, as there will always be FSMs whose implementation requires more memory than is available in the state-of-the-art programmable devices.

In case when efficient memory utilization is essential, the FSM can be implemented in a structure that includes an address register and ROM memory, in which the reduction of ROM memory size is obtained by the introduction of an additional block for address modification (Fig. 8).

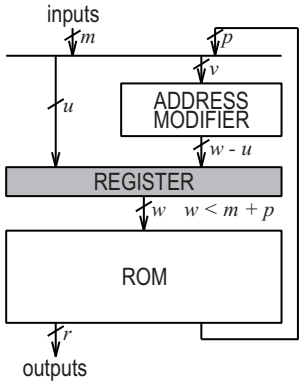

Fig. 8. FSM implementation with the addition of an address modifier

The address modifier can be synthesized with advanced algorithms of functional decomposition, applied until recently exclusively to synthesis of combinational circuits. Such an

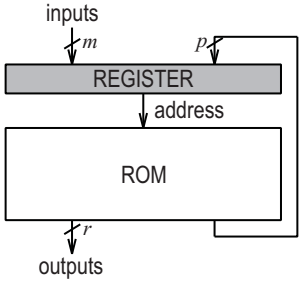

Fig. 9. FSM implementation using ROM memory

TABLE I
FSM TRANSITION TABLE AND $\mathbb{T}$ MAPPING

x_1, x_2	V 00	01	11	10
S	v_1	v_2	v_3	v_4
s_1	s_1	s_4	-	s_2
s_2	-	s_4	s_5	-
s_3	s_3	s_3	s_1	s_2
s_4	s_2	s_1	s_4	-
s_5	s_3	s_2	s_4	s_1

x_1, x_2	V 00	01	11	10
S	v_1	v_2	v_3	v_4
s_1	1	2	-	3
s_2	-	4	5	-
s_3	6	7	8	9
s_4	10	11	12	-
s_5	13	14	15	16

approach to address modifier synthesis was proposed in [25], [24] (and extended in [14], [15]).

The implementation of an FSM shown in Fig. 8 can be seen as a serial decomposition of the memory block included in the structure of Fig. 9 into two blocks: an address modifier and a memory block of smaller capacity than required for the realization of the structure of Fig. 9. As a result, sequential circuits requiring large-capacity ROM memories (and thus not implementable in the architecture of Fig. 9) can be implemented using a memory block with a smaller number of inputs and an additional combinational logic block – the address modifier.

Assuming FSM implementation with FPGA device, the advantage of the proposed architecture lies in that the address modifier can mapped into a network of LUT cells or into a PAL matrix, while the memory block can be mapped into the built-in EAB matrices. The application of this concept (without the optimization of the state encoding) to the synthesis of the earlier discussed benchmark tbk results in a design composed of 333 logic cells and a 4096-bit embedded memory block, which fits entirely in the limited resources of the FLEX structure.

The promising results of other design experiments reported in [24] confirm the effectiveness of the architecture of Fig. 8. The results of the subsequent studies in this area are presented in [14] and [15].

1) Example.: Partition description and partition algebra introduced in [26] are applied to describe logic dependencies in such an FSM.

Based on [26], for the FSM and function $\mathbb{T}$ shown in Table I, the characteristic partition is:

$$P_c = \{\overline{1,8,11,16};\overline{3,9,10,14};\overline{6,7,13};\overline{2,4,12,15};\overline{5}\}.$$

The partition

$$P_1 = \{\overline{1,2,3,4,5,6,7,8,9};\overline{10,11,12,13,14,15,16}\}$$

is related to the partition $\pi = \{\overline{s_1,s_2,s_3};\overline{s_4,s_5}\}$, while the partition

$$P_2 = \{\overline{1,2,4,6,7};\overline{10,11,13,14};\overline{3,5,8,9};\overline{12,15,16}\}$$

is related to the set $\{\pi,\theta\}$, and $\theta = \{\overline{v_1,v_2};\overline{v_3,v_4}\}$.

For set $\mathbb{T} = \{1,2,3,4,5,6\}$ and partitions $P_1 = \{\overline{1,2,5};\overline{3,4,6}\}$, $P_2 = \{\overline{1,2};\overline{3,6};\overline{4};\overline{5}\}$, the quotient partition is:

$$P_1|P_2 = \{\overline{(1,2)(5)};\overline{(3,6)(4)}\}.$$

Let partitions in Π correspond to the state variables and partitions in Θ correspond to the input variables. If $\Pi = \{\pi_1,\ldots,\pi_p\}$ is the set of two-block partitions on S and $\Theta = \{\theta_1,\ldots,\theta_m\}$ is the set of two-block partitions on V, while P_k is a partition on the set T which is related to either π_i or θ_j, then $\mathfrak{p} = \{P_1,\ldots,P_{m+p}\}$ is the set of all partitions related to partitions $\{\pi_1,\ldots,\pi_p,\theta_1,\ldots,\theta_m\}$.

Theorem. To achieve unambiguous encoding of address variables and, at the same time, maintain the consistency relation $\mathbb{T}$ with the transition function, two-block partitions $\mathfrak{P} = \{P_1,\ldots,P_w\}$ have to be found, such that:

$$P_1 \cdot P_2 \cdot \ldots \cdot P_w \leqslant P_c. \tag{19}$$

Although some of the partitions for the $\mathfrak{P}$ set can be selected from the $\mathfrak{p}$ set, the selection is made in such a way that the simplest addressing unit (address modifier) is produced. Such a selection is possible thanks to the method of [15], based on the notion of r-admissibility, [26].

The encoding of state variables can be obtained using the method of construction and coloring of weighted graphs, [15].

Assume that u partitions $\{\pi_1,\ldots,\pi_l\}$ and $\{\theta_1,\ldots,\theta_{u-l}\}$ were chosen. These partitions correspond to the address lines driven by a single variable, either a state variable q or an external variable x. The result is the state and input symbol partial encoding; e.g.,

$$a_1 = q_1,\ldots,a_l = q_l, a_{l+1} = \theta_1,\ldots,a_u = \theta_{u-l}.$$

Corollary. Inequality (19) can be written as:

$$P_{i_1} \cdot P_{i_2} \cdot \ldots \cdot P_{i_u} \cdot P_{i_{u+1}} \cdot \ldots \cdot P_{i_w} \leqslant P_c, \tag{20}$$

where $P_U = P_{i_1} \cdot P_{i_2} \cdot \ldots \cdot P_{i_u}$ is related to the partitions $\{\pi_1,\pi_2,\ldots,\pi_l,\theta_1,\theta_2,\ldots,\theta_{u-l}\}$.

The encoding of the state variables remaining after the partial encoding (input variables, in general) can be obtained from the following rules:

$$\pi_1 \cdot \pi_2 \cdot \ldots \cdot \pi_l \cdot \pi = \pi(\mathbf{0}),\ \theta_1 \cdot \theta_2 \cdot \ldots \cdot \theta_{u-l} \cdot \theta = \theta(\mathbf{0}),$$

where π and θ represent partitions corresponding to these remaining state variables. $\pi(\mathbf{0})$ as well as $\theta(\mathbf{0})$ are partitions whose blocks are equal to their elements.

Inequality (20) can be transformed into:

$$P_U \cdot P_G \leqslant P_c. \tag{21}$$

Corollary. A partition P_G has to be constructed, such that:

$$P_G \geqslant P_V, \tag{22}$$

where $P_G = P_{i_{u+1}} \cdot \ldots \cdot P_{i_w}$ and P_V is related to the partition set $\{\pi, \theta\}$.

Partition P_V can be constructed in the following way:

$$P_V = P_S \cdot P_{V_\theta}, \tag{23}$$

where P_S is the partition related to $\pi(\mathbf{0})$ on the set of states S, and P_{V_θ} is the partition related to θ.

The triple $\langle P_V, E_1, P_1 \rangle$ – where P_1 is a relation representing incompatibilities in quotient partition $P_U|P_c$ on the set T and E_1 is the set of pairs in the relation P_1 – is a multi-graph $M_1(P_V, E_1, P_1)$. Relation of incompatibility in quotient partition $P_U|P_c$ is a relation among all elements in each block of the partition separately.

The coloring of the M_1 multi-graph determines the P_G partition.

The value of

$$\mu = |U| + \lceil \log_2(\chi(M_1)) \rceil \tag{24}$$

determines the size of the required memory, where $\chi(M_1)$ denotes the chromatic number of the M_1 multi-graph.

The size of the required memory is equal

$$M = 2^\mu \cdot (r + p). \tag{25}$$

In the case of $\mu > w$, a new partition P'_V can be constructed. Then, P_V has to be multiplied by appropriately chosen two-block partitions related to those which are generated by input variables from the U set. In that case the result is a non-disjoint decomposition, [15].

Then, the size of the required memory is equal

$$M = 2^w \cdot (r + p). \tag{26}$$

VIII. Experiment Results

As it was stated FSM-based statecharts are transformed into FSM KISS format. The number of resulting FSM elements (i.e. states and transitions) grows exponentially. Table II presents syntax properties of the controllers both statecharts and FSMs. The controller called ***ReaWW*** is described in detail in section 4. The statechart diagram of this controller covers nearly all main syntax feature of statechart formalism. Equivalent FSM consists of 33 transitions and 263 states. Two first controllers, respectively, *demo1* and *demo2*, correspond to diagrams from figures 1 and 4.

TABLE II
Syntax Properties of Controllers

	statecharts							*FSMs*	
name	*#st*	*#tr*	*#seq aut*	*#aut whist*	*#hier depth*	*#in*	*#out*	*#st*	*#tr*
demo1	8	6	4	0	3	1	1	8	9
demo2	3	2	1	0	1	1	4	5	6
Garage	8	7	3	0	2	6	3	14	49
TVrm	8	8	4	1	3	8	5	12	55
Reactor	20	19	8	3	3	10	15	137	986
ReaWW	22	15	9	0	4	10	9	33	263

where abbreviations are: *st* – states, *tr* – transitions, *seq aut* – sequential automata, *aut whist* – automata with history, *hier depth* – hierarhy depth, *in* – inputs, *out* – outputs

Table III presents the results of FSM synthesis according to the algorithm described in section 5. Block *G* corresponds to block *ADDRESS MODIFIER* from figure 8, and block *H* corresponds to block called *ROM*. The number of bits of the memory blocks (*#bit*) is calculated according to formula:

$$\#bit = 2^{\#in} \cdot \#out, \tag{27}$$

where *#in* is the number of address bits and *#out* is the length of the memory word.

TABLE III
Result of Decomposition as a FSM

	block G				*block H*			
name	*#in*	*#out*	*#cube*	*#bit*	*#in*	*#out*	*#cube*	*#bit*
demo1	4	3	7	48	3	4	7	32
demo2	3	2	5	16	2	6	4	24
Garage	9	2	38	1024	6	10	27	640
TVrm	10	4	36	4096	8	11	45	2816
Reactor	16	5	433	327680	11	27	534	55296
ReaWW	13	2	57	16384	9	19	87	9728

Table IV compares results of the synthesis from before ROM-based synthesis (*before*) and after synthesis (*after*). Implementation *before* ROM-based synthesis is realized according to idea from figure 9 and the number of memory bits (*#bit*) is calculated as follows:

$$\#bit = 2^{(\#in + \lceil \log_2(\#tr) \rceil)} \cdot (\#out + \lceil \log_2(\#tr) \rceil), \tag{28}$$

where *#in* and *#out* are, respectively, number of inputs and outputs to the controller and *#tr* is the number of transitions in FSM. Number of bits (*#bits*) *after* synthesis is a simple sum of memory bits of blocks *G* and *H* from table III. The gain in memory bits is calculated according to the following formula:

$$gain = \frac{\#bit_{before} - \#bit_{after}}{\#bit_{before}} \cdot 100\%. \tag{29}$$

Obtained gain reaches more than 90%, especially in complex examples(!). The gain is so huge that decrease in memory bits is better to express by *decreasing ratio*:

$$decreasing\ ratio = \frac{\#bit_{before}}{\#bit_{after}}. \tag{30}$$

The more complicated controller, the decrease in memory bits is bigger and is even tenfold or more!

TABLE IV
Comparision: Encoded FSM Before Decomposition and After

name	*#in*	*#out*	*#q*	*#cube*	*before #bit*	*after #bit*	*gain %*	*dec. ratio*
demo1	1	1	3	9	64	80	-25.00	0.8
demo2	1	4	3	6	112	40	64.29	2.8
Garage	6	3	4	49	7168	1664	76.79	4.3
TVrm	8	5	4	55	36864	6912	81.25	5.3
Reactor	10	15	8	986	6029312	382976	93.65	15.7
ReaWW	10	9	6	263	983040	26112	97.34	37.6

IX. Conclusion

The idea of statechart diagram synthesis through FSM transformation presented above lies in the decomposition of the combinational section of the FSM into two modules: an address modifier and a ROM memory. In general, it is possible to view the address modifier and the memory as separate combinational blocks and implement them independently, applying different strategies for decomposition of these two components. In particular, an alternating application of serial and parallel decomposition has been shown to be an effective strategy to design a structure with both logic cells and EMBs.

Presented synthesis method seems to be very attractive yet another implementation scheme in modern programmable structures equipped with EMBs. Obtained results shows that the method is especially efficient in case of complex controllers.

References

[1] Adamski M. and Węgrzyn M., "Design of reconfigurable logic controllers from petri net-based specifications," in *Discrete-Event System Design - DESDes '09 : preprints of the 4th IFAC Workshop*, University of Valencia, University of Zielona Góra. Gandia Beach, Hiszpania: [B. m.], 2009, pp. 233–238.

[2] G. Borowik, T. Łuba, and P. Tomaszewicz, "A notion of r-admissibility and its application in logic synthesis," in *Discrete-Event System Design - DESDes '09 : preprints of the 4th IFAC Workshop*, University of Valencia, University of Zielona Góra. Gandia Beach, Hiszpania: [B. m.], 2009, pp. 233–238.

[3] R. Wiśniewski, *Syntheis of Compositional Microprogram Control Units for Programmable Devices*. Zielona Góra: University of Zielona Góra Press, Poland, Nov. 2009.

[4] D. Harel, "Statecharts: A Visual Formalism for Complex Systems," *Science of Computer Programming*, vol. 8, pp. 231–274, 1987.

[5] *Unified Modeling Language Specification. Version 1.4.2. ISO/IEC 19501*, Object Management Group, OMG, 250 First Avenue, Needham, MA 02494, U.S.A., Apr. 2005. [Online]. Available: http://www.omg.org/cgi-bin/doc?formal/05-04-01

[6] D. Drusinsky and D. Harel, "Using Statecharts for Hardware Description and Synthesis." *IEEE Transaction on Coputer-Aided Design*, vol. 8, no. 7, pp. 798–807, Jul. 1989.

[7] D. Drusinsky-Yoresh, "A State Assignment Procedure for Single-Block Implementation of State Chart." *IEEE Transaction on Coputer-Aided Design*, vol. 10, no. 12, pp. 1569–1576, Dec. 1991.

[8] S. Ramesh, "Efficient Translation of Statecharts to Hardware Circuits." in *Proceedings of Twelfth International Conference On VLSI Design*, Jan. 1999, pp. 384–389.

[9] *STATEMATE Magnum Code Generation Guide.*, I-Logix Inc., 3 Riverside Drive, Andover, MA 01810 U.S.A., 2001.

[10] K. Buchenrieder, A. Pyttel, and C. Veith, "Mapping statechart models onto an FPGA-based ASIP architecture." in *Proc. EURO-DAC '96*, Sep. 1996, pp. 184–189.

[11] G. Łabiak, "HiCoS Homepage," http://www.uz.zgora.pl/˜glabiak, 2004. [Online]. Available: http://www.uz.zgora.pl/˜glabiak

[12] ——, "From UML statecharts to FPGA - the HiCoS approach," in *Proceedings of Forum on specification & Design Languages – FDL'03*, Frankfurt am Main, Sep. 2003, pp. 354–363.

[13] ——, "From statecharts to FSM-description - transformation by means of symbolic methods." in *Discrete-Event System Design - DESDes '06. A proceedings volume from the 3rd IFAC Workshop*, Rydzyna n. Leszno, Oct. 2006, pp. 161–166.

[14] G. Borowik, B. J. Falkowski, and T. Łuba, "Cost-efficient synthesis for sequential circuits implemented using embedded memory blocks of fpga's," *10th IEEE Workshop on Design and Diagnostics of Electronic Circuits and Systems*, pp. 99–104, 2007.

[15] G. Borowik, "Improved state encoding for fsm implementation in fpga structures with embedded memory blocks," *Electronics and Telecommunications Quarterly*, vol. 54, no. 1, pp. 9–28, 2008.

[16] M. von der Beeck, "A Comparison of Statecharts Variants," in *Proc. of Formal Techniques in real-Time and Fault-Tolerant Systems, Third International Symposium*, ser. LNCS. Springer-Verlag, Sep. 1994, pp. 128–148.

[17] M. Adamski, "Parallel Controller Implementation using Standard PLD Software." in *FPGAs*, W. Moore and W. Luk, Eds. Abingdon EE&CS Books, Oct. 1991, pp. 296–304.

[18] G. Łabiak, *The use of hierarchical model of concurrent automaton in digital controller design, in polish, ISBN: 83-89712-42-3*, ser. Prace Naukowe z Automatyki i Informatyki. Zielona Góra: Oficyna Wydawnicza Uniwersytetu Zielonogórskiego, 2005, vol. VI.

[19] E. Sentovich, K. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, P. Stephan, R. Brayton, , and A. Sangiovanni-Vincentelli, "SIS: A System for Sequential Circuit Synthesis," Department of Electrical Engineering and Computer Science, University of California, Berkeley, Department of Electrical Engineering and Computer Science, University of California, Berkeley, CA 94720, Tech. Rep. UCB/ERL M92/41, May 1992.

[20] F. Somenzi, "CUDD: CU Decision Diagram Package Release 2.4.0." WWW, Feb. 2004, department of Electrical and Computer Engineering University of Colorado at Boulder. [Online]. Available: http://vlsi.colorado.edu/ fabio/CUDD/cuddIntro.html

[21] G. Łabiak, "Symbolic States Exploration of UML Statecharts for Hardware Description," in *Design of Embedded Control Systems*, M. A. Adamski, A. Karatkevich, and M. Węgrzyn, Eds. Springer, 2005, pp. 73–83.

[22] J. E. Hopcroft and J. D. Ullman, *Introduction to Automata Theory, Languages and Computation*. Reading MA: WAddison-Wesley, 2000.

[23] T. Łuba, G. Borowik, and A. Kraśniewski, "Synthesis of finite state machines for implementation with programmable structures," *Electronics and Telecommunications Quarterly*, vol. 55, no. 2, 2009.

[24] M. Rawski, H. Selvaraj, and T. Łuba, "An application of functional decomposition in rom-based fsm implementation in fpga devices," *Journal of Systems Architecture*, vol. 51, pp. 424–434, 2005.

[25] H. Selvaraj, M. Rawski, and T. Łuba, "FSM Implementation in Embedded Memory Blocks of Programmable Logic Devices Using Functional Decomposition," in *Proc. of International Conference on Information Technology: Coding and Computing*, Las Vegas, Apr. 2002, pp. 355–360.

[26] J. A. Brzozowski and T. Łuba, "Decomposition of boolean functions specified by cubes," *Journal of Multi-Valued Logic & Soft Computing*, vol. 9, pp. 377–417, 2003.