PROCEEDINGS

# 55. IWK

Internationales Wissenschaftliches Kolloquium
International Scientific Colloquium

13 - 17 September 2010

# Crossing Borders within the ABC

## Automation,

## Biomedical Engineering and

## Computer Science

Faculty of
Computer Science and Automation

www.tu-ilmenau.de

TECHNISCHE UNIVERSITÄT
ILMENAU

**Impressum**
**Published by**

| | |
|---|---|
| Publisher: | Rector of the Ilmenau University of Technology<br>Univ.-Prof. Dr. rer. nat. habil. Dr. h. c. Prof. h. c. Peter Scharff |
| Editor: | Marketing Department (Phone: +49 3677 69-2520)<br>Andrea Schneider (conferences@tu-ilmenau.de)<br><br>Faculty of Computer Science and Automation<br>(Phone: +49 3677 69-2860)<br>Univ.-Prof. Dr.-Ing. habil. Jens Haueisen |
| Editorial Deadline: | 20. August 2010 |
| Implementation: | Ilmenau University of Technology<br>Felix Böckelmann<br>Philipp Schmidt |

**USB-Flash-Version.**

| | |
|---|---|
| Publishing House: | Verlag ISLE, Betriebsstätte des ISLE e.V.<br>Werner-von-Siemens-Str. 16<br>98693 Ilmenau |
| Production: | CDA Datenträger Albrechts GmbH, 98529 Suhl/Albrechts |
| Order trough: | Marketing Department (+49 3677 69-2520)<br>Andrea Schneider (conferences@tu-ilmenau.de) |

ISBN: 978-3-938843-53-6 (USB-Flash Version)

**Online-Version:**

| | |
|---|---|
| Publisher: | Universitätsbibliothek Ilmenau<br>ilmedia<br>Postfach 10 05 65<br>98684 Ilmenau |



**Home / Index:**
http://www.db-thueringen.de/servlets/DocumentServlet?id=16739

# SOME COMPUTATIONAL ASPECTS OF STATECHARTS FORMAL MODELING OF REACTIVE SYSTEMS

*Grzegorz Labiak*

University of Zielona Góra,
Computer Eng. & Electronics Dept., Podgórna 50,
65-246 Zielona Góra, POLAND, G.Labiak@iie.uz.zgora.pl

## ABSTRACT

The paper presents a graphical notation for modeling of complex behavior of reactive systems. This notation, called statechart diagrams and invented by David Harel, features state-based description, concurrency and hierarchy. Other very essential characteristic of the diagrams is their very strict and formal definition, which allows to apply formal methods (e.g. based on state space characteristic function). This formal definition makes that, on one hand, statechart diagram can be directly implemented in programmable structure, and, on the other hand, their behavior can be analyzed against deadlock detection or can be transformed into other computational model (e.g. FSM). The paper concentrates on relations between some statechart semantic structures and their influence on the number of computational resources, what is very important for the implementation of formal methods algorithms.

***Index Terms—*** statechart diagrams, logic-based synthesis, memory-based synthesis, computational resources consumption

## 1. INTRODUCTION

In the digital controller modeling with statechart diagrams a selection of statecharts syntactic structures has strong influence on some computational aspects. Depending on the target synthesis method, memory-based or logic-based, these aspects falls into two categories, respectively: memory complexity and logic complexity. Memory complexity consists of global states and transitions between them, whereas logic complexity makes up of logic gates.

## 2. SYNTAX OF STATECHARTS

The statechart diagrams ([1]) 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 applications domains caused that many authors proposed their own syntax and semantics ([2]), sometimes differing significantly. Syntax and semantics presented in this paper are intended for specifying the behavior of binary digital controllers which would satisfy as much as possible the UML norm [3].

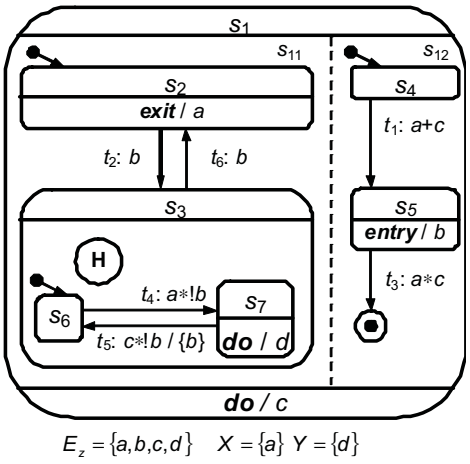


**Fig. 1**. Example of Statechart diagram.

It was assumed that syntax of statecharts (eg author's HiCoS system [4] 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 (makes that state remember its past activity) 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. Others syntax characteristic like cross-level and composite transitions, synch states have been passed over. 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.

## 3. SEMANTICS OF STATECHARTS

For synthesis statechart-base logic controller it is necessary to define precisely its behavior in terms of logic values. In figure 2 a simply diagram and its waveforms illustrate the main dynamics features. Logic value **1** means activity of a state or presence of an event and value **0** means their absence. When transition $t1$ is fired ($T$ = 350) event $t1$ is broadcast and becomes available to the system at next instant of discrete time ($T$ = 450). The activity moves from state *START* to state *ACTION*, where entry action (keyword *entry*) and do-activity (ongoing activity, keyword *do*) are performed (events *entr* and *d* are broadcast). Now transition $t2$ becomes enabled. Its source state is active and predicate imposed on it (event $t1$) is met. So, at the instant of time $T$ = 450, the system transforms activity to the state *STOP*, performs exit action (keyword *exit*, event $ext$) and triggers event $t2$, which do not affect any other transition. The step is finished. 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.

## 4. REACTOR CONTROLLER EXAMPLE

A chemical reactor is an example of industry technological process, in which reacting substances of two kinds are strictly measured out and next are mixed in water environment (Fig. 3). The process consists of three stages:

- water preparing and substrates weighting of given mass – state *FILLING*,
- ingredient stirring in main container for given period of time – state *PROCESS*,
- preliminary process preparing; this stage involves removing discards from scales and from main container – state *INITIATING*.

The operator, who supervises course of process, has at his/her disposal a control desk which is capable of:

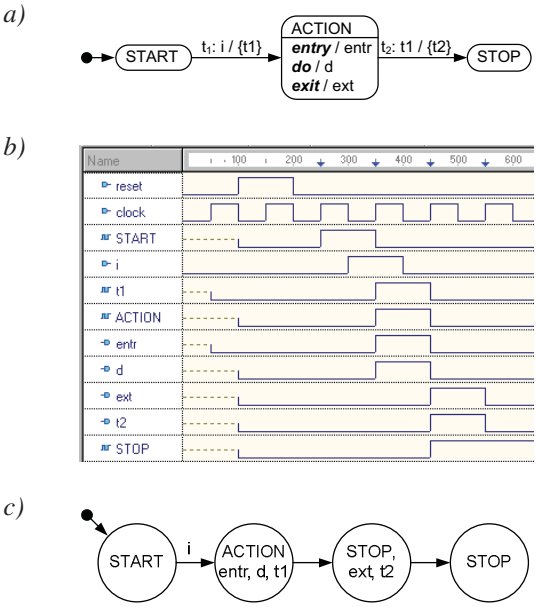


**Fig. 2**. Simple diagram (a), its waveform (b) and equivalent FSM (c)

break-down signalling (*AU* signal), initiate process requesting (*REP* signal), process starting (*AUT* signal). As it is can be seen in the Fig. 3, the operator is allowed to signal break-down in course of filling of containers and in course of chemical process execution. Incoming signals to the controller are signals from weight and level sensors (*B1*, *B2*, *NLIM*, *Nmax*, *Nmin*) deployed in chemical installation and signals from external clocks, which are assigned to measure given time intervals. Outgoing signals from controller are setting signals for pump valves, belt conveyors, mixer engine and for clocks (*V1*, *V2*, *V3*, *V4*, *V5*, *V6*, *EV*, *C1*, *AC1*, *C2*, *AC2*, *M*, *TM1*, *TM2*). Fig. 4 presents reactor block diagram.

Functioning of a chemical plant is as follows (Fig. 3). System starts from state *START* and next in case of lack of break-down signal moves into state *INITIATING*, where main container and belt conveyors are cleared out of previous cycle process remainders. Next, with signal *AUT*, preparing of process ingredient is started

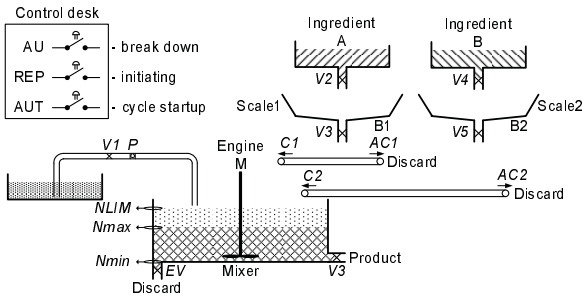


**Fig. 3**. Schematic diagram of the reactor.

– state *FILLING*. In this state, break-down notification (signal *AU*) makes that state *RESTART* becomes active and after the failure is repaired active state of superstate *FILLING* become most recently active ones. Behaviour of this kind is achieved with the use of history attribute. After all the containers (main container and scale containers) are properly filled main chemical process is being started, where reaction time (state *REACTION*) is measured by external clocks. Start of an ensuing process is triggered after signal *AUT* is introduced from control desk, under that condition, that main container is emptied to the desired level (*Nmin* signal). Then system moves to the *FILLING* state.

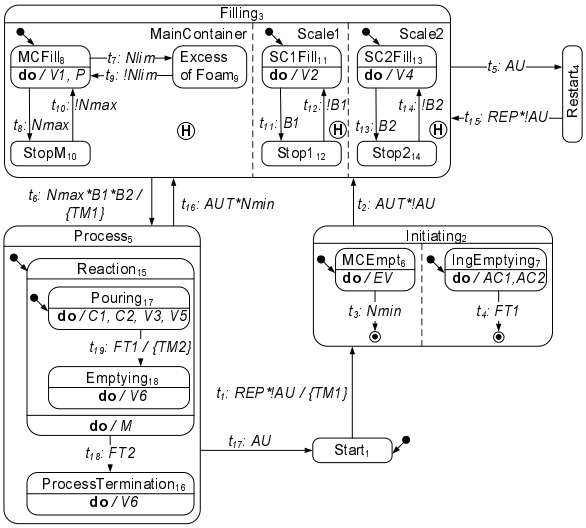


**Fig. 4**. Statechart diagram of the reactor.

## 5. LOGIC-BASED SYNTHESIS

Statechart diagram synthesis is a process where controller described by statecharts is turned into design in terms of logic gates and flip-flops and implemented in hardware (eg. FPGA) [4].

### 5.1. Foundations of hardware implementation

The main assumption of a hardware implementation 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:

- each state is assigned one flip-flop,
- each event is also assigned one flip-flop,
- based on diagram topography and rules of transition executions, excitation functions are created for each flip-flop in a circuit.

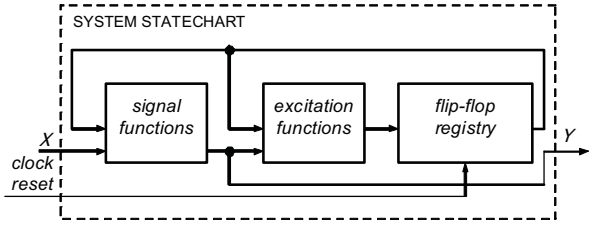


**Fig. 5**. Statechart diagrams Hardware Implementation

Statechart diagrams synthesis yields scheme presented in Fig. 5 and farther description is mainly revolving around specification of flip-flop excitation functions of two types: state flip-flops and event flip-flops.

### 5.2. 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,

**b)** sustaining activity component.

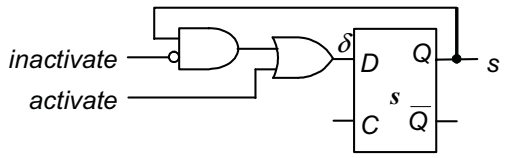


**Fig. 6**. Logic diagram of flip-flop excitation function.

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.

### 5.3. 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:

#### *5.3.1. Transition*

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

$$\delta(e_t) = encond(t) \qquad (3)$$

where $encond(t)$ is fulfilled when both transition source state is active ($active(out(t))$) and control pre-emptive internal condition (regarding *end states*) are fulfilled and events imposed on the transition are present.

#### *5.3.2. 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*):

$$\delta(e_{en}) = \underbrace{\overline{activecond(s)}}_{\text{a}} * \underbrace{\prod_{s_i \in path(root,s)} \delta(s_i)}_{\text{b}} \qquad (4)$$

#### *5.3.3. 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 5.2).

#### *5.3.4. 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*):

$$\delta(e_{ex}) = \underbrace{activecond(s)}_{\text{a}} * \underbrace{\overline{\prod_{s_i \in path(root,s)} \delta(s_i)}}_{\text{b}} \qquad (5)$$

## 6. MEMORY-BASED SYNTHESIS

In some ways statechart-based controller can be perceived as a finite state machine of a Moore type ([5], [6] and see Fig. 2c), so equivalent Moore automaton can be constructed. Transformation of statechart diagrams into FSM model involves constructing equivalent finite state machine, which for external observer behaves just the way statechart does. 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, e.g. KISS format transition table.

Classic Moore automatom is defined as a quintuple $\langle X, S, Y, \delta, \lambda \rangle$, where $X$ is a set of input signals, $S$ is a set of states, $Y$ is a set of output signals, $\delta$ is a transition function and $\lambda$ is an output function dependent only on states. The set of equivalent FSM input signals $X$ is the set of statechart input events, the set of FSM output signals $Y$ is the set of statechart events visible to the environment. The set $S$ of equivalent FSM states is a set of statechart global states which are constructed from local activities: states, actions (transition, *entry*, *exit*) and events broadcast when transition is fired (e.g. $t1$ and $t2$ in diagram from Fig. 2). Transition function $\delta$ is a function which maps current global state into next

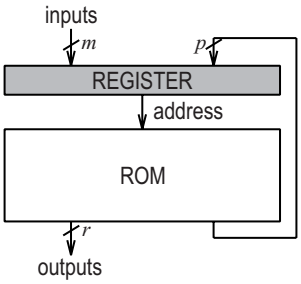


**Fig. 7**. FSM implementation using ROM memory

global state depending on the set of currently accessible events, hence transition function $\delta$ is a vector of Boolean functions. Each component of the vector (in the context of digital synthesis called excitation function) is bound up with either a state or action or transition event. The rules of building this functions are presented in [6] and [4].

FSM can be implemented using memory block (e.g. *Read Only Memory*) and register (see Fig. 7, [7] and [8]). State and input variables constitute memory address variables. The words of memory are storing the encoded present state and output variables. The next state code is formed with input values and present-state code.

Let *m* be the number of inputs, *n* be the number of state variables and *y* be the number of outputs of FSM. Then the size of the memory is equal:

$$M = 2^{(m+n)} \cdot (n + y) \tag{6}$$

where $m+n$ is the size of address, and $n+y$ is the size of the memory word.

Applying functional decomposition the size of the memory block can further be substantially reduced. In this approach memory block is serially decomposed into two blocks: an address modifier and memory block of smaller capacity ([7] and [8]).

## 7. DISCUSSION ON COMPUTATIONAL COMPLEXITY

Statechart diagrams, in comparison with their precursor formalism, i.e. FSM and Petri nets, features quite rich set of syntactic structures. Owing to this syntactic abundance, one behavior can be modeled on may ways. Using various syntactic structure leads to different computational resources consumption, hence it is worth to know how given structure translate into computational resources.

### 7.1. States

Fundamental syntactic feature in statecharts is a state (in the diagrams rounded rectangle) and a signal (which corresponds to event). In terms of logic complexity every state excitation function contributes to regular logic consumption (see equation 2). In terms of memory complexity every state increases number of state variable and number of global states (see equation 6), where increase in number of global state es exponential.

### 7.2. Signals

The increase in input variables, in terms of memory complexity, makes that memory consumption is similar to memory consumption caused be the states. In case of logic synthesis additional signal is yet another variable in Boolean formulas creating logic functions.

### 7.3. Events

The events in the diagrams represent these occurrences which are worth to notice [3], especially are useful in synchronizing overlapping concurrent processes. Every event introduced is as a action (transition, *entry*, *do*, *exit*) is treated like a additional state (except for *do* action which activity is equivalent to state to which is assigned), therefore event increases logic consumption (by its excitation function, see eq. 3, 4 and 5) and increase number of state variables and hence space of global space.

### 7.4. History attibute

The history attribute makes that state remembers its past activity, namely, when activity comes back to compound state, its directly subordinate sequential automaton start its activity not from the start state, but from most recently active state (ie. from this state which was active when activity had been removed by higher hierarchy level transition).

Presence or absence of history attribute makes not much difference to combinatorial logic consumption, it does not affect the number of flip-flops and hence nor the number of state variables, but substantially contributes to increase in number of global states. Equivalent FSM becomes very complex. The difference is clearly visible in *Reactor* controller example. Fig. 4 presents the diagram where history attribute is necessary only for transition $t_5$ and $t_{15}$ (not for $t_2$, $t_6$ and $t_{16}$). When transition $t_6$ is fired compound state *Filling* remembers its past activity (12 possibilities) what in combination with currently active states multiplies the number of global states. This flaw can be remedy by proper synchronization of transition $t_6$, namely this transition is only fired when states *StopM*, *Stop1* and *Stop2* are simultaneously active. This is achieved by using local variable *x*, *y* and *z* (see Fig. 8). By this synchronization the number of global states has been reduced from 137 to only 42 (see Tab. 1 and $Reactor^{\star}$controller).

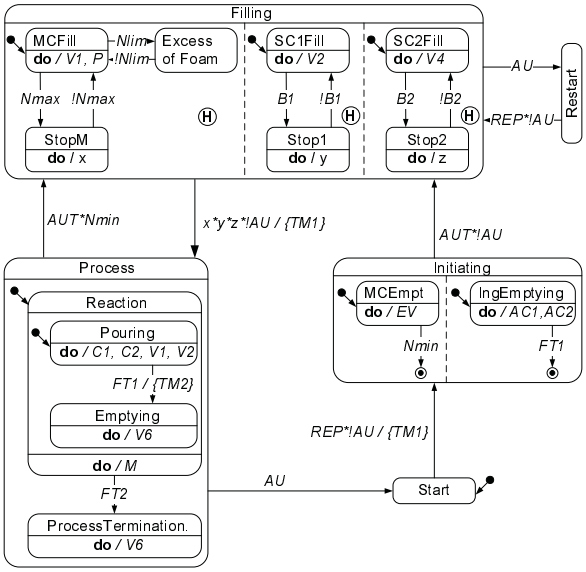


**Fig. 8**. Diagram improved with variables ($Reactor^{\star}$ ).

## 8. CONCLUSION

Statechart diagrams can be synthesized both into programmable structures and memory. In author's approach logic synthesis is conducted through Binary Decision Diagrams (BDDs) which are strongly dependent on variable order. Author's tests, conducted at random variable order, showed that resources consumption takes only small fraction of capacity of modern devices.

Table 1 presents syntax properties of the controllers both statecharts and FSMs. Table 2 compares results of the synthesis from before ROM-based synthesis (*before*) and after synthesis (*after*, realized according to idea presented in section 6). Transformation into FSM features exponential complexity, therefore simplifying input diagram is very crucial. The diagram ($Reactor^{\star}$) is the best example of this.

It is worth to see that synthesis scheme from section 6 gives gain more than 90%, especially in complex examples(!), the more complicated controller, the decrease in memory bits is bigger (tenfold or more!).

**Table 1**. Syntax properties of controllers

| *name* | *#st* | *#tr* | *#seq aut* | *#aut whist* | *#hier depth* | *#in* | *#out* | *FSMs #st* | *FSMs #tr* |
|---|---|---|---|---|---|---|---|---|---|
| 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 |
| Reactor$^{\star}$ | 20 | 19 | 8 | 3 | 3 | 10 | 15 | 42 | 313 |
| ReaWW | 22 | 15 | 9 | 0 | 4 | 10 | 9 | 33 | 263 |

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

**Table 2**. Comparision – encoded FSMs

| *name* | *before #bit* | *after #bit* | *gain %* | *dec. ratio* |
|---|---|---|---|---|
| Garage | 7168 | 1664 | 76.79 | 4.3 |
| TVrm | 36864 | 6912 | 81.25 | 5.3 |
| Reactor | 6029312 | 382976 | 93.65 | 15.7 |
| Reactor$^{\star}$ | 1376256 | 75776 | 94,49 | 18,2 |
| ReaWW | 983040 | 26112 | 97.34 | 37.6 |

## 9. REFERENCES


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

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

[3] Object Management Group, OMG, 250 First Avenue, Needham, MA 02494, U.S.A., *Unified Modeling Language Specification. Version 1.4.2. ISO/IEC 19501*, Apr. 2005.

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

[5] Grzegorz Labiak, "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.

[6] Grzegorz Labiak and Grzegorz Borowik, "Statechart-based controllers synthesis in FPGA structures with embedded array blocks," *Intl Journal of Electronic and Telecommunications*, vol. 56, no. 1, pp. 11–22, 2010.

[7] Mariusz Rawski, Henry Selvaraj, and Tadeusz Luba, "An application of functional decomposition in rom-based fsm implementation in fpga devices," *Journal of Systems Architecture*, vol. 51, pp. 424–434, 2005.

[8] Grzegorz 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.