EVOLVABLE HARDWARE OR LEARNING HARDWARE? INDUCTION OF STATE MACHINES FROM TEMPORAL LOGIC CONSTRAINTS

Marek Perkowski †, Anatoly Chebotarev +, and Alan Mishchenko †, +

† Portland State University,
Dept. of Electr. and Comp. Engn., Portland, Oregon 97207,
Tel: 503-725-5411, Fax: 503-725-4882, *mperkows,alanmi@ee.pdx.edu*

+ Glushkov Institute of Cybernetics, Prosp. Akad. Glushkova 40,
Department of Digital Machines (N 100),
Tel. (38-044) 266-90-48, email: *cheb@d105.icyb.kiev.ua*

Abstract

Evolvable Hardware is Genetic Algorithm (GA) plus reconfigurable hardware. One may ask: "Why **Genetic** Algorithm"? Based on our experience, we question the usefulness of GA as a sole learning method to reconfigure binary FPGAs. Instead, we propose the "Learning Hardware" approach, which consists of creating a sequential network based on feedback from the environment (for instance, positive and negative examples from the trainer), and realizing this network in an array of Field Programmable Gate Arrays (FPGAs).

Here we advocate the approach to Learning Hardware based on Induction of State Machines from Temporal Logic Constraints. The method involves training on examples, constraints solving, determinization, FSM minimization, structural mapping, functional decomposition of multi-valued logic functions and relations, and FPGA mapping. Thus learning occurs on the level of constraints acquisition and functional decomposition rather than on the low level of programming binary switches. Ours is an Occam's Razor learning that allows for generalization and discovery.

Our software algorithms require fast operations on complex logic expressions and solving NP-complete problems such as satisfiability. They should be realized in hardware to obtain the necessary speed-ups. Using a fast prototyping tool, the DEC-PERLE-1 board based on an array of Xilinx FPGAs, we are developing software/configware processors that accelerate the acquisition, synthesis, and optimization of Reactive State Machines.

I. Evolving in Hardware versus Learning in Hardware

In recent years the scientific community is witnessing fast developments in the area of Soft Computing. Thus, Artificial Neural Nets (ANNs), Cellular Neural Nets (CNN), Fuzzy Logic, Rough Sets, Genetic Algorithms (GA), Genetic and Evolutionary Programming, and other approaches have been developed based on learning, adapting, modifying, evolving or emerging. Several mixed approaches are created that in different ways combine elements of these areas with the goal of solving complex and poorly defined problems that could not be tackled by previous, analytic models. What is common to all these approaches is that they propose a way of **automatic learning** by the system; the computer is taught on examples rather than completely programmed (instructed) what to do. This philosophy dominates also the areas of Artificial Life, solving problems by analogy to Nature, decision making, knowledge acquisition, new approaches to intelligent robotics, and many other. **Machine Learning** becomes a new and most general system design paradigm unifying these previously disconnected research areas. It starts to become a **new hardware construction paradigm** as well.

Recently, a term **Evolvable Hardware** (EHW) has been coined [15] which means realization of genetic algorithm (GA) in reconfigurable hardware. It can be exemplified by Brain Builder CBM [15]. The EHW approach to computing raised considerable interest and enthusiasm among some researchers, but scepticism among others. In contrast, our approach of Universal Logic Machine [35, 40, 38, 24, 37, 45] proposes to build a learning machine **based on logic principles**, in particular, Temporal Logic [32, 4, 5, 6, 7], Constructive Induction [2, 11, 27, 28], and Rough Set Theory [34].

While the Genetic Algorithm is a simple and practically blind mechanism of Nature, it can be easily realizable in hardware. We believe that this mechanism alone cannot produce good results. (Although it is relatively easy to do crossover and mutation in hardware, the fitness function evaluation is difficult). In contrast, the logic algorithms that draw upon human knowledge are optimal and mathematically sophisticated. They lead to high quality learning results: knowledge generalization, discovery, no overfitting, small learning errors [47, 1, 26, 20, 21]. Their software realizations, however, use so complex data structures and controls that it is difficult to realize them in hardware.

When we talk about **"Learning Hardware"**, we understand the term "learning" very broadly, as any mechanism that leads to the improvement of operation; evolution-based learning is thus included in it. Although specific learning concepts and their formalisms differ from one learning approach to other, what is common is that, in the process of learning, some kind of (combinational or sequential) network is constructed that stores the knowledge acquired in the learning phase. The learned network is next run on old or new data. The responses may be correct or erroneous. The network's behavior is then evaluated by some fitness (cost) functions and the learning and running phases are alternating.

The process of solving problems consists of two phases: **the phase of learning**, which is, constructing and tuning the network, and **the phase of acting**. The second stage means using knowledge, that is, running the network for data sets. Comparing to the process of developing and using a computer, the first stage could be compared to the entire process of conceptualizing, designing, and optimizing a computer, and the second stage to using this computer to perform calculations. You cannot redesign standard computer hardware, however, when it cannot solve the problem correctly. The Learning Hardware will redesign itself automatically using new learning examples given to it.

II. Logic rather than evolutionary methods for learning

Our ULM approach is based on FPGA technology and associated logic development methods (called **logic synthesis** by the design automation community and **constructive induction** by the Machine Learning community) rather than neural or genetic algorithms. Michie [29] makes distinction between black-box and knowledge-oriented concept learning systems by introducing concepts of **weak** and **strong** criteria. The system satisfies a weak criterium if it uses sample data to generate an updated basis for improved performance on subsequent data. A strong criterion is satisfied if the system communicates concepts learned in symbolic form [28].

Let us observe that ANNs, CNNs, and similar approaches satisfy only the weak criterium while our approach satisfies the strong criterium. We believe that the results of the learning process, and even the process itself, should be rational. They should be similar to those of teaching humans, based on symbolic logic and not on the methods of Nature. Human thinking consists in abstract use of symbols, rather than assigning numeric weights to neurons. Our approach operates on higher and more natural symbolic representation levels. The built-in mathematical optimization techniques (such as graph coloring or satisfiability) support the Occam's Razor Principle, thus solutions found are provably good in the sense of Computational Learning Theory (COLT) [1, 47]. **Thus, learning on symbolic level is the first main point of our approach.**

In our past research, we have been using and comparing, in software, various network structures for learning: two-level AND/OR (Sum-of-Products (SOP), or Disjunctive-Normal-Forms (DNF)) [33], decision trees (C4.5), and multi-level decomposition structures [53, 36, 20, 21], as well as various logic, non-logic and mixed optimization methods: search [37], rule-based, set-covering, graph-coloring, genetic algorithm [18, 16] (including mixtures of logic and GA approaches), genetic programming [17], artificial neural nets, and simulated annealing. We compared the resulting complexity of our networks (Occam's Razor), as well various ways of controlling the number of errors in the learning process [20, 21, 26]. The Decomposed Function Cardinality (DFC) and its extensions for MV logic [1, 20, 21] were used as common measures of complexity, because of their strong theoretically proven properties [1, 47]. Our conclusion, based on these investigations, is that logic approaches and especially the MV decomposition techniques, combined with smart heuristic strategies and good data representations, are usually superior to other approaches due to smaller net complexity and fewer learning errors.

In our experience, especially poor results are obtained using the genetic algorithms [18, 16, 17]. Maybe GA performs well in other applications, but both in our experience and based on literature we could not find a single problem domain in which a GA-based algorithm would be superior to a hand-crafted algorithm to design a binary or multi-valued logic network. This is perhaps because researchers have long experience in creating efficient logic minimization algorithms (for instance, more papers have been written on SOP minimization than perhaps on any other engineering topic). In our approach we want to make use of this accumulated human experience, rather than to "reinvent" algorithms using GA.

III. Learning Hardware Approach in Universal Logic Machine

Developers of evolvable and learning systems agree in that, realized with current software or even parallel programming technologies, the learning phase and/or the execution phase are too slow for real-life problems, especially real-time problems. The situation is essentially the same regardless whether the exhaustive combinatorial search, simulated annealing, or evolutionary algorithms that involve millions of populations are used. Thus, the researchers proposed to speed-up some phases by **migrating from software to hardware.**

Many ambitious projects based on ANNs, cellular logic, DNA, simulated evolution and biologically motivated hardware have been proposed that will be perhaps successful some day, when realized on molecular or quantum levels. However, many of them are quite impractical in **current** technologies.

Most of the approaches to evolvable hardware use **binary Field Programmable Gate Arrays**, because simply there is no other mass-scale hardware reconfigurable (reprogrammable) and relatively inexpensive technology widely available now. Since in binary FPGAs everything is realized on the level of binary logic gates and flip-flops, in our opinion, the learning process should be performed on this level either. **Thus, learning on the level of logic gates and flip-flops is the second main point of our approach.**

We believe that the learning level of sequential logic nets is more natural than the higher level of arithmetic operations of ANNs or Fuzzy Logic functions, or the lower level of routing FPGA connection paths. Once we decide to realize the network using logic gates in FPGA, we should apply efficient logic design algorithms and **realize them in hardware for speedup.** We believe also that all methods that exist in VLSI design, and especially, the powerful EDA (Electronic Design Automation) tools, should be re-used in their entirety, rather than duplicated by naive low-level evolutionary algorithms. Engineers spent many years developing such tools in the area of digital design automation; especially for reconfigurable computers: state machines, logic synthesis, technology mapping, placement and routing, partitioning, timing analysis, etc. Their use will facilitate creation of Learning Hardware. Occam Razor principle should be also used because only it leads to meaningful discoveries and explainable results.

Concluding, we do not believe that the "purist strategies" to evolutionary hardware are practically acceptable for most commercial applications of Learning Hardware. Therefore, we propose the concept of Learning Hardware based on **previous human problem-solving experience** and application of many mathematical algorithms and problem-solving strategies rather than relying on two basic methods of Evolvable Hardware: ANNs and GA. Learning/evolution will remain as the main principle of building new generation hardware, but it should be restricted to higher, abstract levels rather than lowest level FPGA resources. The variant evaluation/selection should be also performed on abstract levels, before mapping to low-level field-programmable resources, for which chromosomes are extremely long and the GA is very inefficient.

The proposed ULM approach to **Learning Hardware** can be summarized as follows:

1. Based on sets of examples specified in our input language **L**, we create a Reactive State Machine (RSM), in particular, a (combinational) function or a relation with no temporal variables. The description consists in input-output specification, initial state specifications and global environment constraints. This machine is usually non-deterministic, but is state-minimal from construction [6, 4, 5, 7]. with respect to all its variables as state variables.

2. The machine can be determinized (converted from non-deterministic to deterministic form). Next the machine is state-minimized (with respect to the new set of state variables, which are a subset of initial input/output variables). We plan also to develop a method to get a state-minimal determinization right away.

3. The machine is mapped to constrained structural resources which we call Regular Automata (RA). It assumes some regular structure, such as a counter, shift register, cellular automaton, or any local-connection based sequential network, and checks for the maximum isomorphism between the autonomous state transition graph of this network and the RSM graph. (The new methods of mapping a state machine to a counter or a shift register are both part of our methodology).

Figure 1: DUAL interface for RSM acquisition and synthesis ("man, wolf, goat and cabbage" problem).

4. The time-based MV logic expressions of Regular Automata are decomposed. We use the algorithm which is our generalization of the functional Ashenhurst-Curtis decomposition [36]. The timed variables, and the multi-valued variables, are converted to new binary variables.

5. The (quasi)optimally constructed network is logically mapped to standard FPGA CLBs and realized using standard partitioning, placement, and routing with the help of EDA tools from Xilinx or other companies. Thus, each RSM is converted to a binary pattern of programming switches in FPGA.

6. The knowledge of the machine is stored in binary memory patterns representing final FPGA reconfigurable information. Under supervision of the software program, the hardware switches between a number of evolved circuits, depending on rules that can also be acquired automatically. This phase is therefore similar to the CBM approach of DeGaris.

7. As the network solves new problems, the new data sets and training decisions are accumulated and the network is repeatedly redesigned. An old network can serve as a redesign plan for a new network, or the latter is "designed from scratch" to avoid any bias.

Thus, we replace the process of creating high level behaviors by evolving on low level used in EHW, with the **ULM model of learning** at high level and next compiling to low level using standard FPGA-based tools. Observe also that the same physical FPGA resources are multiplexed to implement the virtual human-designed **"learning hardware"** and the automatically learned **"data hardware"**. While the "learning hardware" is designed once and cannot be changed, the "data hardware" can be modified indefinitely.

IV. Induction of Reactive State Machines from Temporal Logic Constraints

Because decompositions of combinational logic have been already explained in detail in our previous papers [36, 26, 20, 21], here we concentrate on Reactive State Machines. As explained in the previous section, when the expression (set of examples) in the input language has been created, the problem is closed and becomes that of designing the minimal state machine using regular binary resources of lookup-tables and flip-flops. Our state machine design integrates methods developed in USA and former USSR. We introduce new efficient and practical methods for specification and synthesis of state machines [32, 4, 5, 6, 7] that can significantly improve the solutions currently used in the United States. Developed in the major computer centers of the former Soviet Union, these methods were never published in English or presented to American audience.

Theoretically, any FSM-based EDA tools can be used to solve these tasks. There are, however, some specifics of our approach, which make it different in practice. These differences are related to two issues: (1) The use of temporal logic as the input specification. (2) The use of Regular Automata for structural design.

1. Temporal Logic.

While the temporal logic has been used in the West for verification, there are no papers on its applications in synthesis. At the same time, this logic was used in former USSR for synthesis as well [32, 4, 5, 6, 7]. In addition, to make it compatible with our decomposer, we extend this logic to multi-valued logic data.

In contrast to logic circuit design, the learning data are extremely weakly specified, it is common to have a function or a machine with 99,999% of don't cares. Current EDA software cannot handle such data well, if at all. This fact affects all synthesis algorithms.

Also, we are not familiar with tools that would create a non-deterministic FSM first, and next convert it to a deterministic one. So far, such tools were used only for verification(reachability analysis). Also, state minimization is now an important part of synthesis of **strongly unspecified FSMs**.

2. Regular Automata.

To convert a deterministic or non-deterministic machine to sequential network from CLBs (lookup tables and ff's), instead of a standard state assignment/minimization-factorization/FPGA mapping process, we use the structural synthesis mapping approach based on Regular Automata (RA) design and functional decomposition. Because it is oriented towards lookup-tables, mapping from our RA intermediate format to real chip resources is relatively easy. Good decomposition leads to feature extraction (discovery) and generalization.

This approach is well suited to geometrical constraints imposed on sequential logic, thus it is called the Regular Automata approach. Regular Automata include Cellular Automata as their special case, but every automaton in the array can be different, the connections can be to many nodes, not only to closest geometrical neighbours, etc. More specifically, by Regular Automata we understand sequential netlists placed on a two-dimensional grid in such a way that most of the connections are local, it means, they are to cell's neighbors only, plus there are some global buses and collectors. Every cell may include a number of flip-flops and programmable gates and be connected to two, three, four, eight, twelve, etc neighbors and buses. For instance, there are four neighbors, two vertical and two horizontal buses in the Concurrent Logic Inc. (now ATMEL) architecture [9], which is a special case of our previous model of Regular Layout [49, 50, 43, 41, 42]. Regular Layout is extended to Regular Automata by just adding flip-flops to cells. Flip-flops can be: standard (D, T, JK); and non-standard (half of JK, and START, or logic differentiation [10]). In addition to vertical and horizontal buses, we allow also for diagonal buses. Other examples of Regular Automata are Iterative circuits, Lattice Diagrams, Cellular Automata, counters, shifters, and similar circuits. Practical realizations of Regular Automata are also: the ILU of the CCM [40], and the satisfiability/ESOP minimization machine from [37].

Our methods can preserve partially the structure defined by the original specification and facilitate the design flow that is **transparent** for the designer (graphs of RSMs and autonomous machines are visualized - Fig. 1). The latter is able to supervise the process and influence it when necessary, for example, by changing state encoding or ensuring the proper timing properties. In this way, the designer is not faced with the ready-made results of synthesis as in some of the modern high-level synthesis tools, which mostly rely on incremental synthesis, but is able to quickly produce the effective designs. An interesting aspect is ability to modify the constraints when the designer is not sure of his intention: he experiments with constraints and verifies his ideas on the graphs.

V. The Input Language to Represent the Learning Data

Because we want the system to learn on some higher level than that of elementary gates and their connections, we need first to develop certain higher-level language, in which expressions, the virtual nets, will be automatically created, evaluated, selected and optimized, to be next realized as hardware FPGA nets by top-down automatic design methods. Since in the learning phase we want to operate on elements of this language in hardware, our choices are limited because of the necessity of operations that are easily realizable in hardware.

Several such languages have been created in the past, mostly for applications in Logic Synthesis, Automatic Theorem-Proving, Data Base Theory, and Information Engineering, and we adopted some of them for hardware representation. They include: temporal logic, regular expressions, Petri nets, State Machine tables, tabular representation of data [8], binary and Multi-Valued Cube Calculus [12, 40], Decision Tables, Rough Sets [34], Rough Partitions [25], and Labeled Rough Partitions [22].

	x_1	x_2	y_1	y_2
a	0,2	1	-	2
b	0,1	0	0,2	1
c	2	0	1,2	0
d	1	1	1,2	2

Table 1: Multi-Valued multi-output (combinational) relation in tabular form.

A universal language to specify MV combinational logic is a **two-dimensional tabular representation**, as shown in 1. Rows correspond to objects (examples, samples) a, b, c and d and columns to input variables (attributes) x_1 and x_2, and output variables y_1 and y_2. Symbol y_1 denotes a relation output. Inputs x_1 and x_2 together with output y_1 specify an (oriented) relation. Relation can be used to express such facts as: this color is *red* or *white* but not *yellow* or *black*. Symbol y_2 denotes a function output; $y_2(x_1, x_2)$. Rows c and d have only one value for each attribute, so they are *minterms*. Rows a and b have more than one value for attributes, so they are *cubes*. Eeach row can be thought of as a record from a data base, or their set, or a collection of image features after image preprocessing.

Observe, however, that although such language is quite powerful for Machine Learning, Data Mining and Knowledge Discovery from Data Bases applications, it cannot specify time, it is thus too poor to describe state machines, regular expressions, Petri Nets, path expressions, sequential netlists, grammars and other models, that are used in speech and image recognition, robotics, and other areas. Extension of this language can be done by introducing variables that depend on discrete time. For instance, the example a (row a) from Table 1 can be rewritten to our language as follows:

$$x_1[0,2](t) \ \wedge \ x_2[1](t) \ \Rightarrow y_2[2](t)$$

because both input variables $x_1(t)$, $x_2(t)$ and the output variable $y_2(t)$ are defined in the same moment of time. By allowing previous or next ticks of time, for instance:

$$x_1[0,2](t-2) \ \wedge \ x_2[1](t-3) \ \Rightarrow y_2[2](t+3)$$

we can specify arbitrary regular grammars, regular expressions, sequential netlists, or state machines with multi-valued inputs and outputs. Timed binary variables are next internally converted to standard binary variables, for instance, $x(t) => x$, $x(t-1) => x'$, $x(t-2) => x''$, etc. Similarly, timed MV variables are internally converted to standard binary variables. Various binary encodings of MV symbols can be used.

To understand the power of our language, we described successfully problems, such as "cannibals and missionairies", "man, wolf, goat and cabbage", "dining philosophers", models of Pavlov's behaviors, various real-time protocols, and other problems specified by temporal logic constraints. Based on them, we can appreciate that only two simple generalizations of MV logic: (1) adding time moments to logic variables, (2) allowing to formulate equations not only in explicit way (the output variables as the functions of the input variables) but also as non-explicit, convoluted MV equations involving input, output, and auxiliary (state, constraint) variables, allow us to create a language of a high expressive power and also quite natural. The language allows also to specify temporal and logic constraints, such as "the man cannot be lazy, he has to move the animal or the item to another shore of the river", "cannibals cannot row", or "only three people can be in the boat").

Let us consider one example of temporal logic specification used in our methodology. This is the well-known problem of the man, the wolf, the goat, and the cabbage. At the beginning, all of them are on the left bank of the river, and the man should transport them to the right bank. The wolf and the goat, as well as the goat and the cabbage, cannot be left alone on the same bank without man. The boat is navigated by the man, who can take only one object with him.

All the boolean variables are first-order predicates depending on discrete time. $M(t)$ is true when the man in the left bank and false when the man in on the right bank. The same applies to variables $W(t), G(t), C(t)$, which denote the wolf, the goat, and the cabbage respectively.

Here is the temporal logic specification of constraints for this problem.

1) The wolf and the goat cannot stay on the left bank and on the right bank without the man:
$(W(t) \& G(t)) \Rightarrow M(t)$. $(\sim W(t) \& \sim G(t)) \Rightarrow \sim M(t)$.

The goat and the cabbage cannot stay on the left bank and on the right bank without the man:
$(G(t) \& C(t)) \Rightarrow M(t)$. $(\sim G(t) \& \sim C(t)) \Rightarrow \sim M(t)$.

2) If the wolf is on the left (right) bank, it means that either the wolf stayed there or the man has brought it there from the right (left) bank one unit of time before:
$W(t) \Rightarrow (W(t-1) \mid \sim W(t-1) \& \sim M(t-1) \& M(t))$. $\sim W(t) \Rightarrow (\sim W(t-1) \mid W(t-1) \& M(t-1) \& \sim M(t))$.

The same for the goat and the cabbage:
$(G(t) \Rightarrow (G(t-1) \mid \sim G(t-1) \& \sim M(t-1) \& M(t))$. $\sim G(t) \Rightarrow (\sim G(t-1) \mid G(t-1) \& M(t-1) \& \sim M(t))$.
$(C(t) \Rightarrow (C(t-1) \mid \sim C(t-1) \& \sim M(t-1) \& M(t))$. $\sim C(t) \Rightarrow (\sim C(t-1) \mid C(t-1) \& M(t-1) \& \sim M(t)\ )$.

3) Any two objects cannot be brought across the river at the same time:
$(W(t) \Leftrightarrow W(t-1)) \mid (G(t) \Leftrightarrow G(t-1))$.
$(W(t) \Leftrightarrow W(t-1)) \mid (C(t) \Leftrightarrow C(t-1))$.
$(G(t) \Leftrightarrow G(t-1)) \mid (C(t) \Leftrightarrow C(t-1))$.

4) The man is not lazy:
$M(t-1) \Leftrightarrow \sim M(t)$.

Here the symbols $\sim$, $\&$, $\mid$, $\Rightarrow$, and $\Leftrightarrow$ designate complemented variable, logic AND,logic OR, logic implication $(\neg a \vee b)$, and logic equivalence $(a \equiv b)$, respectively.

The result of synthesis from this specification using our software system DUAL is shown in Figure 1. The state transition graph given in dotted lines corresponds to the non-deterministic algorithm, which the man should follow to safely transport all the objects to the right bank. This algorithm consists of seven steps, with two possible variations, depending on whether he chooses to take the cabbage or the wolf when the goat is already trasported to the right bank. Observe that the designer (the supervisor, the environment), who gives the language **L** expressions to the learning system, does not know the state machine, he only specifies constraints on the behavior of the system or sample sequences of input and output values.

VI. DEC-PERLE-1 Board for Fast Prototyping

The DEC-PERLE-1 board [52] is organized around a central computational matrix made up of 16 Xilinx XC3090 LCAs[1], surrounded by a four 1MB RAM banks, and 7 other LCAs to implement switching and controlling functions. We modeled our algorithms in software. We also realized the first version of ULM, called Cube Calculus Machine (CCM) on DEC-PERLE [37, 38, 24]. These were very useful learning experiences. The DEC-PERLE-1 board, similarly to other FPGA boards, advocates **regular design styles** without many control signals. It is then good for small SIMD processors, pipelining, systolic processors, Cellular Automata or complex (decomposed) Boolean functions. The basic design principle is: **"map two-dimensional tables to two-dimensional logic resource arrays"**, leading to the introduction of the concept of **Regular Automata**. The design can be developed incrementally thanks to its easy memory access, host interface with FIFOs, and the clock debugging modes and tags. Based on these principles, we developed next a new variant of the Cube Calculus Machine and evaluated its speed-up. The goal is now to map software algorithms such as binate covering used in FSM minimization or unate covering used in decomposition to efficient hardware algorithms. This can be done by checking satisfiability [35, 45].

VII. Demonstration of Learning Hardware in Robotics, or "An Improved Furby"

Similarly to DeGaris and his ROBOKONEKO robot-kitten, we need some real-life environment to demonstrate the practicality of our ideas. We have chosen the humanoid arms/head robot: the software/hardware machine connected to an inexpensive robot (OWI kits) and a solid-state camera.

Mechanically, the robot will be build from two OWI robotic arms as two hands, and the modified OWI arm to play the role of a head. More motors and gears from Robix and LEGO Masterminds will be used for eyes, brows, and ears movements. Thus an upper body of a human puppet will be assembled, controlled by a total of $3 * 5 + 4 * 2 = 23$ motors. There will be also several sensors on each hand and the face: temperature, touch, infrared, and other. The signals from them will be converted in A/D converters to multi-valued input signals of the reactive state machines.

The robot will perform simple programmable movements such as: **MOTOR-1-UP, MOTOR-2-DOWN, MOTOR-3-LEFT, PICK, RELEASE**. They correspond to states of output variables. For each OVI arm there are 5 motors, each in 3 states (left, right, none). All signals are programmed as multi-valued. Timing information is added.

At first, we follow the standard "learning from supervisor" approach, and the human is the supervisor. He creates all sequences and constraints for the reactive state machine. It is as if the parent would teach the child by **directly re-wiring his brain** using positive and negative examples. Thus, learning is done with the human as the feedback loop. The set of sequences is incomplete, so the machine performs the **generalization** automatically. Adding or removing new rules, by the human supervisor or automatically/randomly, will change the behavior. What obstacles can the robot's arm encounter? How to react to them? How the robot should react to or mimique the human's behaviors seen by the camera? A lengthy process of trial and error is necessary to answer these questions. The goal of this project is to design the system that reacts to sound, temperature, touch, words (text) from speech recognition [51], simplified image recognition [39], light sensors, etc.

We want to design something like the **famous Furby toy, but with real learning.** Let us first discuss how Furby works. It can be observed that Furby's internal states are prespecified, its learning is only transiting to prespecified new states in the labyrinth of its possible "states of emotions and knowledge" (sleeps, plays games, sings, is ill/healthy, etc). Appropriate learning patterns (such as petting the head twice and next the back once) lead to the displays of appropriate behaviors pre-stored in the toy's memory, and are only hidden from the user by not entering some internal states earlier. Although this is not a real learning, it is perceived as astounding by people who observe this toy. Now, we want to have a toy similar in sensor/actuator Pavlovian/Skinnerian learning model, but capable of building its own "world model" and

[1] LCA stands for Logic Cell Arrays

\ *Approach to* *Aspect of* \ *"Learning Hardware"* *Comparison.*\	Brain Builder	Universal Logic Machine
Model of Learning	Artificial Neural Net	Reactive State Machine
Training Data	Sample Vectors	Multi-Valued Temporal Logic Constraint Language **L**
How the net is constructed	Genetic Algorithm, ANN Training	Reactive State Machines construction and minimization, Multi-Valued Logic Synthesis
Virtual intermediate representation	Cellular Automata (CA)	Regular Automata (RA)
What is learned:	CA tables	Binary sequential logic nets
Net construction realized in:	Hardware (intrinsic EHW)	Hardware and software (software-hardware codesign)
Mapped to:	Array of binary FPGAs	Array of binary FPGAs
Hardware platform:	Xilinx 6000 series CBM	Xilinx 3090 + on-board memory DEC-PERLE-1 board + DEC workstation

Table 2: Brain Builder (De Garis) versus Universal Logic Machine.

internal model with unlimited behaviors. The new states corresponding to changed behavior will be created using our RSM approach.

The demonstration illustrates an approach to programming robots based on learning from examples. Robot control units (i.e. RSMs) are constructed as a hierarchy of behavior modules concurrently running in multi-tasking environment. As the result, the robot will exhibit simple actions to which more sophisticated behavior patterns can be added by adding higher-level control. This way of organizing a hierarchy of control modules is known as subsumption architecture. This is a concrete way of introducing multi-tasking and concepts of real-time control to electrical engineering students as it is done in MIT, CMU etc. What is new about our approach, is that the RSM is synthesized (evolved) automatically from input/output relationships specified in our concurrent constraint language.

VIII. Conclusion

Within the "Learning Hardware" general philospphy, we have been pursuing the approach that we call the Universal Logic Machine (ULM). Table 1 shows the comparison of the Brain Builder CAM-Brain Machine (CBM) and the ULM approaches.

We presented an overview of our past and current research as well as the goals of the entire ULM project. If paper were accepted, our presentation will include demonstration of the RSM software.

References

[1] Y. Abu-Mostafa (ed.), " Complexity in Information Theory," *Springer Verlag,* New York, 1988, p. 184.

[2] R.L. Ashenhurst, "The Decomposition of Switching Functions", *Proc. Int. Symp. of Th. of Switching,* 1957.

[3] R.A. Brooks, "A Robust Layered Control System for a Mobile Robot," *IEEE Journal of Robotics and Automation,* 2(1):14, 23,March 1986.

[4] A.N. Chebotarev, "Approach to Functional Specification of Automata Systems," *Kibernetika i Sistiemnyj Analiz,* 1991, No. 3, pp. 31-42, (in Russian).

[5] A.N. Chebotarev, "Consistency Test for Simple Specification of Automata," *Kibernetika i Sistiemnyj Analiz,* 1994, No. 3, pp. 3-14, (in Russian).

[6] A.N. Chebotarev, "Synthesis of a Nondeterministic Automaton from its Logical Specification,I," *Kibernetika i Sistiemnyj Analiz,* 1995, No. 5, pp. 3-15, (in Russian).

[7] A.N. Chebotarev, "Synthesis of a Procedural Representation of an Automaton Specified in the Logical Language **L***," *Kibernetika i Sistiemnyj Analiz,* 1997, No. 4, pp. 60-74, (in Russian).

[8] E.F. Codd, "A Relational Model of Data for Large Shared Data Banks," *Comm. ACM*, 13, pp. 377-387.

[9] Concurrent Logic Inc., "CLI 6000 Series Field Programmable Gate Arrays," *Preliminary Information*, Dec. 1, 1991, Rev. 1.3.

[10] M. A. Perkowski, A. Coppola, "A State Machine PLD and Associated Minimization Algorithms," *Proc. of the FPGA'92, 1992 ACM/SIGDA First Intern. Workshop on Field-Programmable Gate Arrays,* pp. 109 - 114, Berkeley, February 16-18, 1992.

[11] H.A. Curtis, "A New Approach to the Design of Switching Circuits," *Princeton,* N.J., Van Nostrand, 1962.

[12] D.L. Dietmeyer, "Logic Design of Digital Systems," *Allyn and Bacon, Boston, MA,* 1971.

[13] H. DeGaris, "Evolvable Hardware: Genetic Programming of a Darwin Machine," In "Artificial Nets and Genetic Algorithms," R.F. Albrecht, C.R. Reeves and N.C. Steele (eds), *Springer Verlag,* pp. 441-449, 1993.

[14] *http : //www.hip.atr.co.jp/ ∼ degaris*

[15] H. DeGaris, "Evolvable Hardware: Principles and Practice," *CACM Journal,* August 1997.

[16] K. Dill, and M. Perkowski, "Minimization of Generalized Reed-Muller Forms with Genetic Operators," *Proc. Genetic Programming '97 Conf.,* July 1997, Stanford Univ., CA.

[17] K. Dill, J. Herzog, and M. Perkowski, "Genetic Programming and its Application to the Synthesis of Digital Logic," *Proc. PACRIM '97,* Canada, August 20-22, 1997.

[18] K. Dill, and M. Perkowski, "Evolutionary Minimization of Generalized Reed-Muller Forms," *Proc. ICCIMA'98 Conference,* pp. 727-733, February 1998, Australia, published by World Scientific.

[19] M. Domsch, "MIT 6.270 LEGO Robot Design Competition," World Wide Web, URL ishttp://www.mit.edu/courses/6.270/home.html.

[20] C. Files, M. Perkowski, "An Error Reducing Approach to Machine Learning Using Multi-Valued Functional Decomposition," *Proc. ISMVL'98,* pp. 167 - 172, May 1998.

[21] C. Files, M. Perkowski, "Multi-Valued Functional Decomposition as a Machine Learning Method," *Proc. ISMVL'98,* pp. 173 - 178, May 1998.

[22] S. Grygiel, and M. Perkowski, "New Compact Representation of Multiple-Valued Functions, Relations, and Non-deterministic State Machines," *Proc. ICCD'98,* October 1998.

[23] T. Higuchi, M. Iwata, and W. Liu (eds), "Evolvable Systems: From Biology to Hardware," *Lecture Notes in Computer Science,* No. 1259, *Proc. First Intern. Conf. ICES'96,* Tsukuba, Japan, October 1996, Springer Verlag, 1997.

[24] L. Jozwiak, M.A. Perkowski, D. Foote, "Massively Parallel Structures of Specialized Reconfigurable Cellular Processors for Fast Symbolic Computations," *Proc. MPCS'98 - The Third International Conference on Massively Parallel Computing Systems,* Colorado Springs, Colorado - USA, April 6-9, 1998.

[25] T. Luba, J. Rybnik, "Algorithmic Approach to Discernibility Function with Respect to Attributes and Object Reduction," *Int. Workshop on Rough Sets,* Poznan 1992.

[26] R. Malvi, M. Perkowski, and L. Jozwiak, "Exact Graph Coloring for Functional Decomposition: Do we Need it?," pp. 1-10, *Proceedings of 3rd International Workshop on Boolean Problems,* Freiberg University of Mining and Technology, Institute of Computer Science, September 17-18, 1998.

[27] R.S. Michalski and J.B. Larson, "Inductive Inference of vl Decision Rules," in *Workshop in Pattern-Directed Inference Systems,* Hawaii, May 1977.

[28] R.S. Michalski, I. Bratko, and M. Kubat, "Machine Learning and Data Mining: Methods and Applications," *Wiley and Sons,* 1998.

[29] D. Michie, "Machine Learning in the Next Five Years," *Proc. EWSL'88, 3rd European Working Session on Learning,* Glasgow, *Pitman,* London, 1988.

[30] A.A. Mishchenko, "A CAD System for Automated Synthesis of Controlling Automata," *Cybernetics and System Analysis,* Plenum Press, 1997, No. 3, pp. 23-30.

[31] A.A. Mishchenko, "On Properties of Reversible Polynomial Functions," *Cybernetics and System Analysis,* Plenum Press, 1997, No. 4, pp. 44-49.

[32] M.K. Morokhovets, A.N. Chebotarev, "Resolution Approach to Testing Compatibility of Interacting Automata," *Kibernetika i Sistemnyj Analiz,* 1994, No. 6, pp. 36-50, (in Russian).

[33] L. Nguyen, M. Perkowski, N. Goldstein, "PALMINI - Fast Boolean Minimizer for Personal Computers," *Proc. of the IEEE/ACM 24th Design Automation Conference,* pp. 615 - 621, Miami, Florida, June 28 - July 1, 1987.

[34] Z. Pawlak, "Rough Sets. Theoretical Aspects of Reasoning about Data," *Kluwer Academic Publishers,* 1991.

[35] M. Perkowski, "Systolic Architecture for the Logic Design Machine," *Proc. of the IEEE and ACM International Conference on Computer Aided Design - ICCAD'85,* pp. 133 - 135, Santa Clara, 19 - 21 November 1985.

[36] M. Perkowski, M. Marek-Sadowska, L. Jozwiak, T. Luba, S. Grygiel, M. Nowicka, R. Malvi, Z. Wang, and J. S. Zhang, "Decomposition of Multiple-Valued Relations," *Proc. ISMVL'97,* Halifax, Nova Scotia, Canada, May 1997, pp. 13 - 18.

[37] M. Perkowski, P. Lech, Y. Khateeb, R. Yazdi, and K. Regupathy, "Software-Hardware Codesign Approach to Generalized Zakrevskij Staircase Method for Exact Solutions of Arbitrary Canonical and Non-Canonical Expressions in Galois Logic," *Booklet of 6th Intern. Workshop on Post-Binary ULSI Systems,* Nova Scotia, Canada, May 27, 1997, pp. 41 - 44.

[38] M. A. Perkowski, L. Jozwiak, and D. Foote, "Architecture of a Programmable FPGA Coprocessor for Constructive Induction Approach to Machine Learning and other Discrete Optimization Problems", in Reiner W. Hartenstein and Victor K. Prasanna (ed) *"Reconfigurable Architectures. High Performance by Configware,"* IT Press Verlag, Bruchsal, Germany, 1997, pp. 33 - 40.

[39] M. Perkowski, S. Wang, W.K. Spiller, A. Legate, E. Pierzchala, "Ovulo-Computer: Application of Image Processing and Recognition to Mucus Ferning Patterns," *Proc. of the Third IEEE Symposium on Computer-Based Medical Systems,* pp. 52 - 59, Chapel Hill, North Carolina, June 3-6, 1990.

[40] M.A. Perkowski, "A Universal Logic Machine," an invited address, *Proc. of the 22nd IEEE International Symposium on Multiple Valued Logic, ISMVL'92,* pp. 262 - 271, Sendai, Japan, May 27-29, 1992.

[41] M.A. Perkowski, E. Pierzchala, and R. Drechsler, "Ternary and Quaternary Lattice Diagrams for Linearly-Independent Logic, Multiple-Valued Logic and Analog Synthesis, *Proc. ISIC-97,* Singapur, 10-12 Sept.1997.

[42] M. Perkowski, E. Pierzchala, and R. Drechsler, "Layout-Driven Synthesis for Submicron Technology: Mapping Expansions to Regular Lattices," *Proc. First Intern. Conf. on Inf., Comm. and Signal Processing, ICICS'97*, Singapur, 9-12 Sept. 1997. Session 1C1: Spectral Techniques and Decision Diagrams.

[43] M.A. Perkowski, M. Chrzanowska-Jeske, and Y. Xu, "Lattice Diagrams Using Reed-Muller Logic," *Proc. RM'97 Conference,* Oxford Univ., U.K., Sept. 1997, pp. 85 - 102.

[44] M. Perkowski, L. Jozwiak, and S. Mohamed, "New Approach to Learning Noisy Boolean Functions," *Proc. ICCIMA'98 Conference,* February 1998, Australia, published by World Scientific, pp. 693 - 706. Australia, published by World Scientific.

[45] M. Perkowski, "Do It Yourself Reconfigurable Supercomputer that Learns," *book preprint*, Portland, Oregon, 1999.

[46] PSU POLO Directory with DM/ML Benchmarks, software and papers: *http : //www.ee.pdx.edu/polo/*

[47] T. D. Ross, M.J. Noviskey, T.N. Taylor, and D.A. Gadd, "Pattern Theory: An Engineering Paradigm for Algorithm Design," *Final Technical Report WL-TR-91-1060,* Wright Laboratories, USAF, WL/AART/WPAFB, OH 45433-6543, August 1991.

[48] P. Sapiecha, M. A. Perkowski, and T. Luba, "Decomposition of Information Systems Based on Graph Coloring Heuristics," *Symposium on Modelling, Analysis and Simulation,* CESA'96 IMACS Multiconference, Lille, France, July 9-12, 1996.

[49] A. Sarabi, N. Song, M. Chrzanowska-Jeske, M. A. Perkowski, "A Comprehensive Approach to Logic Synthesis and Physical Design for Two-Dimensional Logic Arrays," *Proc. DAC '94,* San Diego, June 1994, pp. 321 - 326.

[50] N. Song, M.A. Perkowski, M. Chrzanowska-Jeske, A. Sarabi, "A New Design Methodology for Two-Dimensional Logic Arrays," *VLSI Design,* 1995, Vol. 3., Nos. 3-4, pp. 315-332.

[51] K.B. Stanton, P.R. Sherman, M.L. Rohwedder, Ch.P. Fleskes, D. Gray, D.T. Minh, C. Espinosa, D. Mayi, M. Ishaque, M.A. Perkowski, "PSUBOT - A Voice-Controlled Wheelchair for the Handicapped," *Proc. of the 33rd Midwest Symp. on Circuits and Systems,* pp. 669 - 672, Alberta, Canada, August 1990.

[52] J. Vuillemin, P. Bertin, D. Roncin, M. Shand, H. Touati, and Ph. Boucard, "Programmable Active Memories: Reconfigurable Systems Come of Age," *IEEE Trans. on VLSI Systems,* Vol. 4, No. 1., pp. 56-69, March 1996

[53] W. Wan, and M. Perkowski, "A New Approach to the Decomposition of Incompletely Specified Multi-Output Function Based on Graph Coloring and Local Transformations and Its Application to FPGA Mapping," *Proc. Euro-DAC*, pp. 230 - 235, 1992.