Specification and Synthesis of Parallel Hierarchical Finite State Machines for Control Applications

Valery Sklyarov, Iouliia Skliarova

Abstract—**Many practical algorithms require support for hierarchy and parallelism. Hierarchy assumes an opportunity to activate one sub-algorithm from another and parallelism enables different sub-algorithms to be executed at the same time. The paper presents a graphical specification of parallel hierarchical algorithms, suggests architecture of a parallel reconfigurable controller, indicates limitations and describes a formal method of synthesis allowing the given algorithms to be implemented in hardware on the basis of the proposed architecture.**

I. Introduction

FIG. 1 presents architecture of a control system for which the presented technique might be helpful. The paper covers all the steps needed for specification and synthesis of a digital controller shown in a dashed rectangle. Primary objective is to design high-performance controllers, which are useful for such complex control systems that are considered in [1,2].

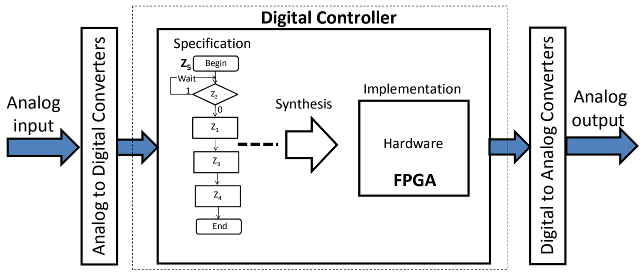

Fig. 1. Basic structure of considered control systems.

It should be noted that besides of control systems, specification and synthesis of finite state machines (FSMs) is required for vast variety of practical applications. For many of them FSMs have to provide support for hierarchy and parallelism [3-11].

Let us consider some examples. Suppose it is necessary to describe the functionality of a computational system implementing a set of operations where some operations involve the execution of other operations. Thus, it would be very desirable that the relevant control unit would provide support for modularity, hierarchy and reusability of the previously described operations. To model such a control unit by an FSM the latter has to support the relevant specifications.

Let us consider another example, which is simple enough to demonstrate the use of the considered technique. Suppose we need a priority buffer (a priority queue) for control (or some other) devices (see Fig. 2). Note that such buffers are quite common for numerous embedded applications.

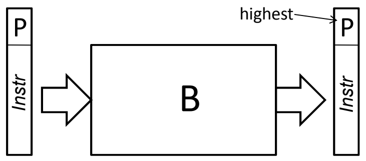

Fig. 2. General structure of a priority buffer B.

The buffer B receives instructions on its input. Each instruction contains a code (*Instr*) of the required operation and a field *P* indicating the priority of the instruction. On external request the buffer has to output the instruction with the highest priority. Note that incoming instructions alter the sequence of priorities that has been previously established. Thus, the buffer must provide a flexibly reconfigurable sequence of instructions. Let us implement the described functionality with two micro-programs (MP1, MP2) that execute the following operations of the relevant control unit: MP1 – constructs in the buffer memory a special binary tree whose nodes are associated with the incoming instructions; MP2 – extracts the instruction with the highest priority and deletes the extracted instruction from the tree. Since incoming and outcoming instructions have to be handled concurrently, MP1 and MP2 have to execute in parallel. The paper suggests a specification for hierarchical and parallel algorithms and the synthesis of parallel hierarchical FSM (PHFSM) from the proposed specifications.

PHFSMs are more sophisticated models than conventional FSMs but they are more adequate for many practical applications [1-11]. Examples demonstrating advantages of PHFSM were presented: in [3] for computational systems; in [4] for multi-agent interaction protocols; in [5] for solving mathematical problems; in [6] for control applications; in [7] for interest management, in [8] for robotics; in [10] for Ethernet access chip; and in [12,13] for embedded systems.

V. Sklyarov is with University of Aveiro/IEETA, 3810-193 Portugal (phone: +351234401539; fax: +351234370545; e-mail: skl@ua.pt).
I. Skliarova, is with University of Aveiro/IEETA, 3810-193 Portugal (e-mail: iouliia@ua.pt).

There are many different approaches enabling parallel specifications to be implemented in hardware. A frequently used model is a set of communicating FSMs [9,10]. Each of them is responsible for a specific operation, and different operations can be performed at the same time, i.e. several FSMs can be active simultaneously. There are also some other approaches, such as FSMs, based on compatible partial states [4], mapping parallel algorithms applying one-hot state encoding technique [12], converting parallel specifications (such as UML) to synthesizable hardware description language (HDL) [11], etc. Practically all known methods permit to design either a parallel or a hierarchical FSMs and do not allow combining hierarchy and parallelism within the same device.

The approach considered here differs from other related work in this area in several ways. Firstly, it explicitly states some of pre-defined constraints (see section II), which allows complex verification procedures to be avoided. Secondly, the PHFSM model is described with the aid of a customizable and reusable HDL template (see section III). Thirdly, the specification is synthesizable (see section IV) in the sense that it permits customizable parts of the HDL template to be configured, thus enabling a complete synthesizable HDL specification to be built and afterwards implemented in hardware.

The remainder of this paper is organized in five sections. Section II presents the proposed specification of parallel and hierarchical algorithms and gives an example. Section III describes the structure and HDL template of PHFSM. Section IV discusses a method of PHFSM synthesis. Section V summarizes the implementation details and the results of experiments. The conclusion is given in Section VI.

II. Specification of Parallel Hierarchical Finite State Machines

To describe the functionality of a PHFSM, different forms of behavioral specifications [3,6] can be applied. This paper suggests parallel hierarchical specification based on the previously developed hierarchical graph-schemes (HGS) extended to implement parallel micro-programs (let us call such an HGS a parallel HGS, or simply PHGS).

PHGSs have the following formal description based on HGSs [14]. A PHGS is a directed connected graph containing rectangular, rhomboidal and triangular nodes. Each PHGS has one entry point, which is a rectangular node named *Begin* (see Fig. 3, a), and one exit point, which is a rectangular node named *End* (see Fig. 3, b). Rectangular nodes contain either a micro-instruction or a macro-instruction, or both (see Fig. 3, c). A micro-instruction includes a subset of micro-operations from the set $Y=\{y_1,...,y_N\}$. A micro-operation is an output signal, which causes a simple action in the execution unit. A macro-instruction incorporates a subset of macro-operations from the set $Z=\{z_1,...,z_Q\}$. Each macro-operation is described by another PHGS of a lower level. Parallelism is provided when a macro-instruction includes more than one macro-operation. Each rhomboidal node contains one element from the set X, where $X=\{x_1,...,x_L\}$ is the set of logic conditions (see Fig. 3, d). A logic condition is an input signal, which communicates the result of a test. Each triangular node contains an expression which can produce a set of one-hot values associated with the outputs of this node (see Fig. 3, e). When the control flow passes a triangular node, exactly one output must be selected enabling the control flow to proceed. Directed lines (arcs) connect the inputs and outputs of the nodes in the same manner as for an ordinary graph-scheme [14]. Let us assume that $Z(\Gamma_h)$ is the set of macro-operations that belong to the PHGS Γ_h. If $Z(\Gamma_h)=\varnothing$ we have an ordinary graph-scheme [14].

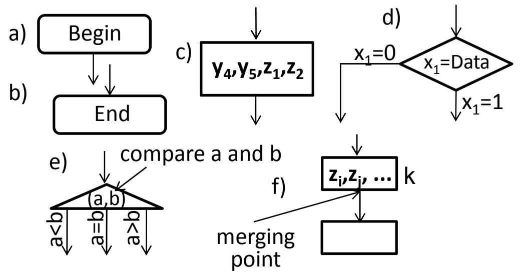

Fig. 3. Nodes of a PHGS.

PHGSs enable to develop any complex control algorithm step by step, concentrating efforts at each stage on a specified level of abstraction (that is, on a particular element of the set Z). Each component of the set Z is usually relatively simple, and can be checked and tested independently. In addition, any component can be updated and reused.

The following remarks have to be taken into account and the following constraints have to be satisfied for the proposed specification:

- The output of a rectangular node k with more than one element $z_i,z_j,\ldots$ from the set Z is called a merging point (see Fig. 3, f). Control flow passes the merging point if and only if all the elements $z_i,z_j,\ldots$ have been completed. This means that a node following the node k is only activated after terminating all macro-operations $z_i,z_j,\ldots$;
- One macro-operation z_i can affect the execution of another macro-operation z_j through synchronization flags (semaphores) in such a way that z_i sets a flag when it starts and z_j checks the flag making it possible to conclude if it can be executed or not;
- Although recursion is allowed, there exists the following limitation: any recursive macro-operation cannot include parallel activation of other macro-operations.
- If macro-operations $Z_k=\{z_i,z_j,\ldots\}$ are running in parallel, then for any $z_i \in Z_k$ activation of another macro-

operation $z_j \in Z_k$ that is already running in parallel is not allowed. In other words the basic rule is the following: the maximum number of macro-operations running in parallel must be known before execution of any PHGS. This permits to make PHGSs synthesizable in hardware. On the other hand recursion is allowed, i.e. $z_i \in Z_k$ can activate $z_i \in Z_k$ once again. Obviously, the general rules that apply to recursive procedures, known in software [15] and hardware [16] engineering, have to be satisfied, i.e. after a certain number of iterations there must be a non recursive exit that allows the recursive macro-operation to end;

- In accordance with [14], the execution of an HGS is synchronous and the execution of PHGSs is synchronous as well.

Fig. 4,5 present an example of PHGSs describing the functionality of the priority buffer shown in Fig. 2 and $Z = \{z_0, z_1, z_2, \ldots, z_6\}$ (z_0 corresponds to the top level algorithm). There are also a number of micro operations and logic conditions, however not all of them are shown in Fig. 4, 5. The PHGSs for the elements of Z contain the following macro-operations: z_1 – receive and add data; z_2 – extract and remove data; z_3 – synchronize and add data; z_4 – find and send data with the highest priority; z_5 – synchronize and remove data; z_6 – remove extracted data. Rectangular nodes with “Wait” are empty and they permit for testing of some logic conditions in each clock cycle (this will be explained in more detail later). PHGSs z_3 and z_5 in Fig. 5 demonstrate a possible simplification for “Wait” nodes.

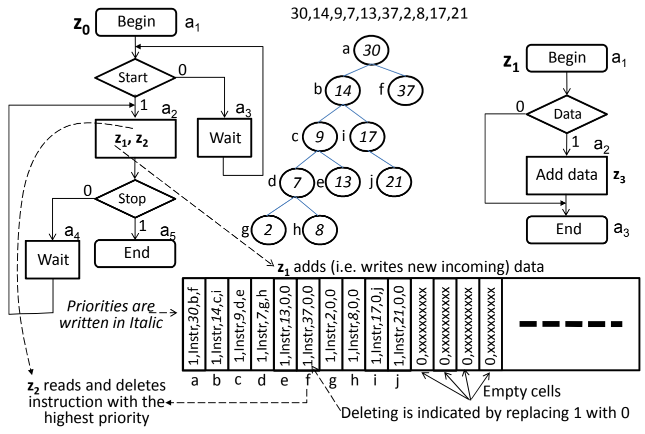

Fig. 4. Examples of PHGSs z_0 and z_1 for priority buffer from Fig. 2.

PHGS z_0 in Fig. 4 executes two macro-operations z_1 and z_2 in parallel (z_1 hierarchically calls z_3, and z_2 calls z_4 and z_5). The macro-operations z_3 and z_6 are recursive. The maximum number of PHGSs running in parallel is 2.

The required functionality of the priority buffer is achieved using the following method [16]. PHGS z_1 receives incoming data and with the aid of z_3 constructs a binary tree (shown as an example in Fig. 4) whose nodes contain five fields that are: a pointer to the right child node, a pointer to the left child node, an instruction code (*Instr*), a priority value, and an indicator, which is set to 1 in any used for a node memory cell and reset to 0 as soon as a node is removed from the tree. The nodes are maintained so that at any node, the left sub-tree contains only priority values that are smaller than the priority value at the node, and the right sub-tree contains only priority values that are greater. It is assumed that different instructions cannot have the same priority. Thus, the priority value for each node is unique. To build such a tree, we have to find the appropriate place for each incoming node in the current tree. In order to extract instruction with the highest priority, we can use known algorithms [16,17] and one such algorithm has been implemented in z_4. PHGS z_6 removes extracted data and is very similar to z_4.

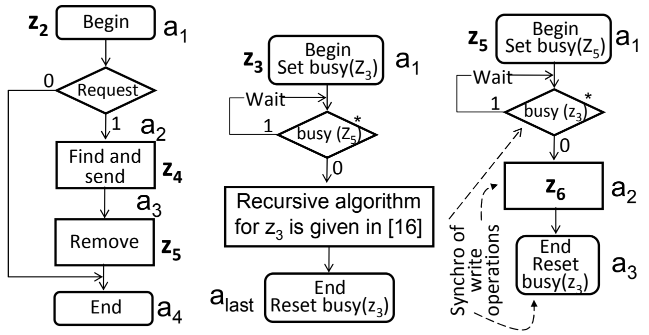

Fig. 5. Examples of PHGSs z_2, z_3 and z_5 for buffer from Fig. 2.

PHGS z_1 receives and adds data to the binary tree, which is incrementally constructed in memory. The construction mechanism is shown in Fig. 4 for an incoming sequence of instructions with the priorities: 30,14,9,7,13,37,2,8,17,21. The resultant binary tree is also shown in Fig. 4. The first instruction has the priority 30 and the first memory cell (see the cell a in Fig. 4) is selected. Each cell has an indicator that is set to 1 for a node that is in use and reset to 0 when a node has been removed from the tree allowing the memory to be reused. Thus, the indicator for the first cell is set to 1. Since initially there are no sub-trees the relevant pointers are set to point to nothing (which is coded by some predefined value). The next incoming instruction with the priority 14 has to be placed as the left child node. Thus, a new memory cell is selected and the pointer to the left sub-tree in the first cell is altered appropriately. The indicator for the second cell is set to 1. The same technique is applied to all subsequent instructions. A dual port memory has been used for the buffer and thus, writing (PHGS z_1) and reading (PHGS z_4) can be done at the same time. PHGSs z_3 and z_5 are in parallel branches but they both write to memory cells and therefore must not be executed at the same time. This is achieved through semaphores (see Fig. 5) that suspend one

PHGS when the other PHGS is being executed. Conditions *busy(z_5)* and *busy(z_3)* (in rhomboidal nodes marked with an asterisk (*) in Fig. 5) cannot be equal to 1 at the same time because a priority encoder is used (the signal *Set busy(z_3)* has higher priority than *Set busy(z_5)*).

III. Structure and HDL template for parallel hierarchical finite state machines

It is known that modularity and hierarchy can be implemented in an FSM by replacing its register memory with a stack memory [14] (such a FSM, called a hierarchical FSM (HFSM), is shown in Fig. 6,a). An HFSM permits the execution of HGSs and contains two stacks, one for states (*FSM_stack*) and the other for modules (*M_stack*). The stacks are managed by a combinational circuit (CC) that is responsible for new module invocations and state transitions in any active module that is designated by outputs of the *M_stack* [16]. The stack pointer is common to both stacks.

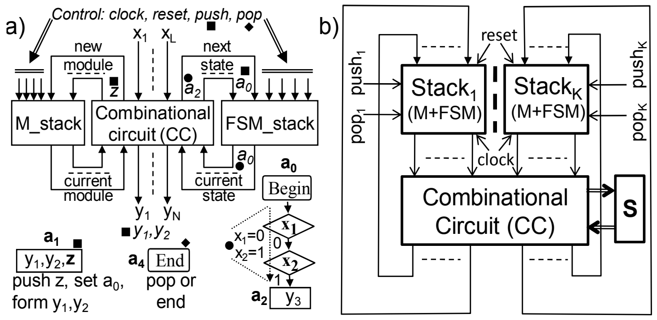

Fig. 6. The structures of HFSM (a) and PHFSM (b).

A PHFSM is considered to be an HFSM with K *M_stacks* and *FSM_stacks* (see Fig. 6,b), where K is the maximum number of macro-operations running in parallel. The functionality of the PHFSM has to be described in such a way that K is known in advance. The block S is composed of semaphores for synchronization of parallel macro-operations. The implementation of merging points is done inside the combinational circuit.

The PHFSM template is a customizable hardware description language code (VHDL has been used as HDL), which can be formally altered for the given set of PHGSs at the initial steps of the synthesis in such a way that: a) it describes the PHFSM which implements the given set of PHGSs, and b) it is synthesizable in commercially available synthesis tools. In general, a PHFSM template is composed of reusable, parameterized stack(s) and a customizable CC. The semaphores are flip-flops that can be set/reset within one macro-operation in order to block/unblock the execution of another macro-operation.

Basic VHDL code for the reusable stacks (stack$_1$,…, stack$_K$) is exactly the same as for the HFSM [16] and it is shown below.

```
process(clock,reset)
begin
  if reset = '1' then  stack_ptr <= 0;
    FSM_stack(stack_ptr) <= a1;
    M_stack(stack_ptr) <= z0; error <= '0';
    return_flag <= '0';
  elsif rising_edge(clock) then
    if push = '1' then
      -- hierarchical call
      if stack_ptr = stack_size then
        error <= '1';
      else stack_ptr <= stack_ptr + 1;
        FSM_stack(stack_ptr+1) <= a1;
        FSM_stack(stack_ptr) <= NS;
        M_stack(stack_ptr+1) <= NM;
      end if;
    elsif pop = '1' then
      -- hierarchical return
      stack_ptr <= stack_ptr - 1;
      return_flag <= '1';
    else -- non-hierarchical state transition
      return_flag <= '0';
      FSM_stack(stack_ptr) <= NS;
    end if;
  end if;
end process;
```

Here `NS` is the next state, `NM` is the next module (macro-operation), `a1` (a_1) is the first state in each module, `z0` (z_0) is the top-level module, described by the top-level PHGS. The signal `return_flag` is needed to allow for analysis of logical conditions that might be changed in a hierarchically called macro-operation. This permits the correct state transition to be selected after a hierarchical return (this will be shown later on examples). Please note that the code above is basic because it does not take into account known optimization techniques such as minimizing time for hierarchical calls/returns, etc. [17].

The CC is a customizable circuit with almost the same structure as the CC for HFSMs [14] (see the code below). The first level of the case statement selects modules and the nested case statements describe state transitions within each module.

```
process (current_module,current_state,inputs)
begin
  case M_stack(stack_ptr) is
    when z0 =>
      case FSM_stack(stack_ptr) is
        -- state transitions in the module z0
        -- generating outputs for the module z0
      end case;
    when z1 =>
      case FSM_stack(stack_ptr) is
        -- state transitions in the module z1
        -- generating outputs for the module z1
      end case;
    -- repeating for all modules
end process;
```

IV. Synthesis of parallel hierarchical finite state machines

Let us consider how to construct the circuit of PHFSM whose outputs depend only on states (i.e. for the Moore model). The proposed method of synthesis includes the following three steps: 1) transforming the PHGSs to a synthesizable PHFSM; 2) customizing the described in the previous section PHFSM template; 3) synthesis of PHFSM from the customized template with the aid of commercially available tools.

The first step assigns PHFSM states to all given PHGSs. It is done by labeling certain PHGS nodes and associating the labels with PHFSM states afterwards. The second step incorporates into the template all transitions between the states required by PHGSs. This enables us to produce a synthesizable VHDL code for the given PHGSs. The last step executes synthesis and constructs a circuit of the PHFSM, which can be physically implemented in hardware, for example in field-programmable gate arrays (FPGA).

Labeling is performed using the following method, which is very similar to [14]. The first label a_1 is assigned to the node *Begin* of all modules z_q (q=1,…,Q). The labels $a_2,a_3…,a_M$ are assigned to unmarked rectangular nodes in each of the PHGSs. Here, M is the maximum number of labels in a PHGS with the maximum number of rectangular nodes. At the next phase, the labels are considered to be PHFSM states. Obviously, different PHGSs might have the same labels and, thus, the same states. The proper module is correctly recognized by the name of the module, which is also provided. Thus, the pair "*the module name q*" and "*the state name a_m*") is sufficient to know which rectangular nodes of PHGSs are being active at any time. Fig. 4 and 5 demonstrate how the given PHGSs have to be labeled.

At the next step all transitions between the states have to be described in the template for CC. There are a total of 5 different types of state transitions, which are described below.

1) Simple sequential state transitions between states within the same module are provided in the same way as in an ordinary FSM, for example the transitions from a_1 in the module z_0 will be coded as follows:

```
case M_stack1(stack_ptr1) is
  when z0 =>
    case FSM_stack1(stack_ptr1) is
      when a1 =>
        if (Start='1') then
           next_state1 <= a2;
         else next_state1 <= a3;
        end if;
```

2) Simple (non parallel) hierarchical transitions, for example, a hierarchical transition from the state a_3 in the module z_2 will be coded as follows (the signal `return_flag` is needed to allow for testing logic conditions, if any, after returning from z_5 and to avoid the second invocation of module z_5 during the return step):

```
when a3 =>
  if return_flag1 = '0' then
    push1 <= '1'; next_state1 <= a3;
    next_module1 <= z5;
  else push1 <= '0'; next_state1 <= a4;
  end if;
```

3) Parallel hierarchical calls, for example, the following call from the state a_2 in the module z_0:

```
when a2 =>
  if return_flag1 = '0' then
     push1 <= '1'; push2 <= '1';
    next_module1 <= z1; next_module2 <= z2;
  else push1 <= '0'; push2 <= '0';
    if (Stop = '1') then next_state1 <= a5;
    else next_state1 <= a4;
    end if;
  end if;
```

4) Hierarchical returns in merging points (i.e. hierarchical returns from parallel branches), for example, the return from the state a_3 in the module z_1:

```
when a3 => next_state1 <= a3;
  if ( (stack_ptr1 > 0) and
       (FSM_stack2(stack_ptr2) = a4) )
    then pop1 <= '1';
  else  pop1 <= '0';
  end if;
```

5) Simple hierarchical returns, which are similar to the previous point but do not require any additional test like "`and(FSM_stack2(stack_ptr2)=a4)`", for example, the return from the state a_3 in the module z_5 (it should be taken into account that for z_5 the second stack is used):

```
when a3 => next_state2 <= a3;
  if (stack_ptr2 > 0)  then pop2 <= '1';
  else                 pop2 <= '0';
  end if;
```

Note that there are two blocks of stacks for the example considered. Each block includes *M_stack* and *FSM_stack* (see Fig. 6). Let us consider the graph in Fig. 7, which permits to calculate the number of modules that can be executed in parallel. All possible parallel modules are indicated by closed dashed curves. The maximum number of parallel modules is equal to the total number of the disjoint innermost closed dashed curves (CDCs). For example, if there are 3 outer CDCs; the first CDC has 2 nested CDCs and one nested CDC has 4 CDCs inside. In this case the number of disjoint innermost CDCs is equal to 2+1+4=7. For our example the maximum number of parallel branches is equal to 2.

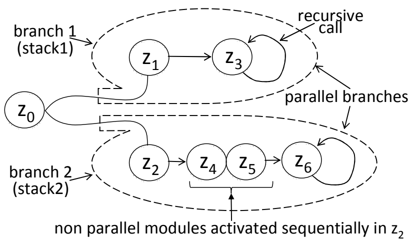

Fig. 7. Parallel and hierarchical calls of different modules.

The graph in Fig. 7 enables us to know the number K of stacks (see Fig. 6) and to distribute modules (PHGSs) between the stacks. Let us assign modules z_0, z_1, z_3 to the $stack_1$ and the remaining modules to the $stack_2$. That is why in the code above all signals associated with stack memories have the relevant indexes (*e.g.* `pop1`, `next_module2`, etc.).

After applying the rules considered above, the template will be customized and the PHFSM can be synthesized from the template.

V. Implementation and experiments

The priority buffer considered above and described by PHGSs in Fig. 4, 5 was implemented and tested in FPGA of Xilinx Spartan-3 family. Synthesis from the VHDL specification was done in Xilinx ISE.

The suggested buffer was incorporated into a system for garage control [18], which was offered as a work for M.Sc. thesis in 2008/2009 academic year. The garage control system is organized as a composition of sub-systems communicating through a wireless (RF) interface. The central sub-system is responsible for the garage control, i.e. for the set of operations such as processing and indication of the most preferable slot for parking of any new car on the entrance; opening/closing the gates, providing instructions for parked cars that have to be retrieved, etc. Other sub-systems are installed inside cars and they instruct cars how to drive to the slots indicated by the central sub-system.

The functionality of the central sub-system relies on a priority buffer. The latter takes input data items about free and released parking slots and outputs items with the highest priority. The buffer is organized in such a way that the established priorities might be dynamically rearranged as well as some items can be removed on external requests (for example, when some parking slots are reserved for special purposes).

Another example where PHFSMs have been used is presented in [12] for a self-controlled transport section.

The primary goal of the experiments was to prove on working examples that the model and the method presented in the paper are correct and effective. Synthesis and implementation results prove that the considered PHFSMs require reasonable FPGA resources and can be constructed on the basis of low-cost FPGAs.

VI. Conclusion

The paper presents methods of specification and synthesis of parallel hierarchical finite state machines that permit to implement such algorithms that: 1) are composed of modules (described by the proposed parallel hierarchical graph-schemes); 2) enable the modules to be activated from other modules; 3) allow more than one module to be activated at the same time. The model combines multiple stack memories interacting with a combinational circuit. The synthesis involves three basic steps: 1) transforming the given parallel hierarchical graph-schemes to a synthesizable PHFSM; 2) customizing the proposed PHFSM template; 3) synthesis of PHFSM from the customized template with the aid of commercially available tools. The results of experiments have shown that the proposed model and methods are very practical and enable the designers to implement real-world hierarchical and/or parallel algorithms.

References

[1] Y.V.Mitrishkin, V.N.Dokuka, R.R.Khayrutdiniov, A.V.Kadurin, "Plasma magnetic robust control in tokamak-reactor", *Proc. 45th IEEE Conf. on Decision and Control*, San Diego, Dec. 13-15, 2006.

[2] M.Ariola, G.Ambrosino, A.Pironry, J.Lister, P.Vyas, "Design and experimental Testing of Robust Multivariable Controller on a Tokamak", *IEEE Trans. on Control Systems Technology*, vol. 10, no 5, 2002, pp. 646-653.

[3] S.E. Lyshevski, Nanocomputer Architectronics and Nanotechnology. In Handbook of Nanoscience, Engineering and Technology, CRC press, 2003.

[4] D. Cheremisinov, L. Cheremisinova, "Developing Agent Interaction Protocols with PRALU", *International Journal Information Theories & Applications*, vol. 13, no. 3, pp. 239-246, 2006.

[5] M. Aldridge, "A Parallel Finite State Machine Implementation of a Nearest-Eight Hoshen-Kopelman Adaptation for Landscape Analysis", *ACMSE 2007*, North Carolina, USA, pp. 391-394, 2007.

[6] A.D. Zakrevskij, Parallel Algorithms of Logic Control, Moscow, URSS, 2003.

[7] J. Shi, N.I. Badler, M.B. Greenwald, "Joining a Real-Time Simulation: Parallel Finite-State Machines and Hierarchical Action Level Methods for Mitigating Lag Time", *Proc. 9th Conference on Computer Generated Forces*, May, 2000.

[8] V. Sklyarov, I. Skliarova, "Hierarchical Specification and Design of Control Systems in Robotics", *Proc. ICARA'2006*, Palmerston North, New Zealand, pp. 623-628 2006.

[9] D. Kim, S. Ha, "Static Analysis and Automatic Code Synthesis of flexible FSM Model", *Proc. ASP-DAC*, pp. 161-165, 2005.

[10] Z. Liu, G. Shi, L. Zeng, "A parallel FSM design method and its application in ten gigabit Ethernet access chip", *Proc. ASIC'2003*, pp. 870-873, 2003.

[11] D. Björklund, J. Lilius, "From UML Behavioral Descriptions to Efficient Synthesizable VHDL", *Proc. 20th IEEE Norchip Conference*, pp. 1-6, 2002.

[12] V. Sklyarov, I. Skliarova, "Design and Implementation of Parallel Hierarchical Finite State Machines", *Proc. HUT-ICCE 2008*, Hoi An, Vietnam, June 4-6, pp. 33-38, 2008.

[13] S.Edwards, L.Lavagno, E.A.Lee, A.Sangiovanny-Vincentelli, "Design of Embedded Systems: Formal Models, Validation, and Synthesis", *Proc. of the IEEE*, vol. 85, no. 3, March, pp. 366-390, 1997.

[14] V. Sklyarov, "Hierarchical Finite-State Machines and Their Use for Digital Control", *IEEE Transactions on VLSI Systems*, vol. 7, no 2, pp. 222-228, 1999.

[15] F.M. Carrano, J.J. Prichard, Data Abstraction and Problem Solving with C++, Addison Wesley, 2002.

[16] V. Sklyarov, "FPGA-based implementation of recursive algorithms", *Microprocessors and Microsystems, Special Issue on FPGAs: Applications and Designs*, vol. 28/5-6, pp. 197-211, 2004.

[17] V. Sklyarov, I. Skliarova, "Recursive and Iterative Algorithms for N-ary Search Problems", *2nd IFIP Symposium on Professional Practice in Artificial Intelligence - AISPP'2006*, Santiago de Chile, Chile, pp. 81-90, 2006.

[18] V. Skliarov, I. Skliarova, A. Neves, "Modeling and Implementation of Automatic System for Garage Control", *Proc. of the ICROS-SICE International Conference 2009*, Fukuoka, Japan, August 2009.