Design and implementation of parallel hierarchical finite state machines

Conference Paper · July 2008

DOI: 10.1109/CCE.2008.4578929 · Source: IEEE Xplore

CITATIONS
20

READS
776

2 authors, including:

Iouliia Skliarova
University of Aveiro
165 PUBLICATIONS **1,096** CITATIONS

SEE PROFILE

Design and Implementation of Parallel Hierarchical Finite State Machines

Valery Sklyarov, Iouliia Skliarova
Department of Electronics, Telecommunications and Informatics/IEETA,
University of Aveiro
3810-193 Aveiro, Portugal
skl@ua.pt, iouliia@ua.pt

***Abstract*—This paper presents a novel model and method for synthesis of parallel hierarchical finite state machines (PHFSM) that permit to implement algorithms composed of modules in such a way that 1) the modules can be activated from other modules, and 2) more than one module can be activated in parallel. The model combines multiple stack memories interacting with a combinational circuit. The synthesis involves three basic steps: 1) conversion of a given specification to special state transition diagrams; 2) use of the proposed hardware description language templates; 3) synthesis of the circuit from the templates. A number of PHFSMs have been designed, implemented in low-cost commercially available FPGAs, tested, and evaluated. The results of experiments have proven the effectiveness and practicability of the proposed technique for solving real-world problems.**

Keywords-parallel and hierarchical algorithms; parallel hierarchical finite state machine; VHDL specification; synthesis; FPGA

I. INTRODUCTION

Finite state machines (FSM) can be seen as fundamental building blocks for vast varieties of digital systems and they are frequently used in computers, embedded controllers, application specific integrated circuits, industrial electronic devices, etc. That is why it is very important to explore common models of FSMs and formal methods of their synthesis. Basically, we can distinguish three types of models, which are *simple sequential*, *hierarchical*, and *parallel*. In turn, they can be further divided (for example, we can consider recursive and iterative hierarchical models), but for the purposes of this paper the presented above types are sufficient and we will examine them with a bit more detail.

Models and methods of synthesis for *simple sequential* FSMs are very well studied [1,2] and they are considered just as a basis for more complicated hierarchical and parallel FSMs.

A *hierarchical* FSM is composed of other hierarchical and simple sequential FSMs (let us call them modules), which can be activated much like procedures in software programs. Thus, any module can be triggered from either another or the same module, which permits to implement the well-known strategy of "divide and conquer". Models and methods of synthesis for hierarchical FSMs are considered in [3].

A *parallel* FSM enables different modules to be executed in parallel. Note that generally any electronic device deals with simultaneous processing of analog/digital signals. Thus, it is parallel by definition. That is why exploring models and methods of synthesis for parallel FSMs is very important and greatly demanded. Some results in this scope are reported in [4,5].

The most interesting approach is combination of parallel and hierarchical capabilities within the same FSM. Let us call such FSMs *parallel hierarchical* FSMs (PHFSM). They were examined briefly in [5] but the conclusion drawn was that it would be very difficult to combine hierarchy and parallelism within the same machine. The presented in this paper model and method demonstrate that such a combination is achievable.

The subsequent sections introduce a novel model of PHFSM and a novel method for synthesis of PHFSM. Their applicability is demonstrated on an example of a simple embedded controller. All the relevant experiments have been done with circuits of PHFSM tested in commercially available FPGAs (Field-Programmable Gate Arrays).

The remainder of the paper is organized in five sections. Section II suggests a novel model of PHFSM, which permits to implement parallel hierarchical specification. Section III presents an example of a simple embedded controller, which will be used to demonstrate applicability of the model. Section IV describes a novel method of PHFSM synthesis based on hardware description language templates. Section V discusses experiments and practicability of the proposed model and method and summarizes the contribution of the paper. The conclusion is given in Section VI.

II. A MODEL OF PARALLEL HIERARCHICAL FINITE STATE MACHINES

Models and behavioral specifications of parallel and hierarchical FSMs are closely related to each other. Although a model might be considered, in a certain sense, as specification independent, it is better to establish first a "model-specification" relationship, which makes it easier to understand which processes are going to be modeled and how it has to be done. Thus, at the beginning, a specification method for parallel hierarchical algorithms is going to be introduced.

It is known that parallel hierarchical algorithms can be described by hierarchical graph-schemes (HGS) [3] with more than one macro-operation to be activated in the same rectangular node (actually in the same FSM state). Since, the latter is the only difference with traditional HGS, their formal definition is skipped (it is the same as in [3]). We will call such HGS parallel hierarchical graph-schemes (PHGS). Fig. 1 depicts an example of PHGS, which describes functionality of a simplified embedded controller (this controller will be introduced in the following section).

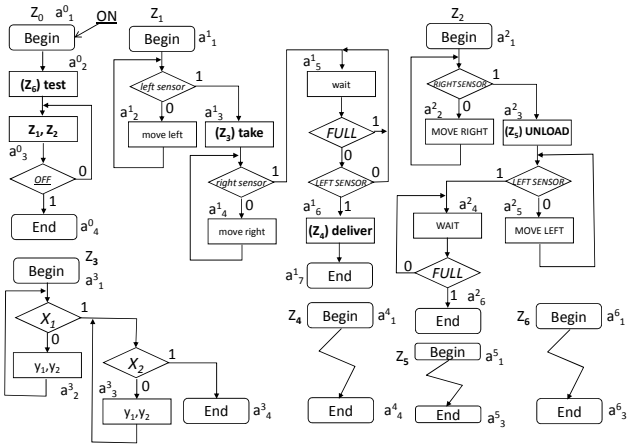

Figure 1. An example of parallel hierarchical algorithms for an embedded controller

The algorithm is composed of 7 modules $Z_0,\ldots,Z_6$. Some of these modules, namely $Z_1,\ldots,Z_6$, are activated hierarchically and some of them, namely Z_1,Z_2, are called in parallel. Labels like a^0_1 and a^0_2 represent states and we will introduce them a bit later. Rhomboidal nodes contain logical conditions that are formed by sensors of the embedded controller and enable the sequence of execution of the algorithm to be properly selected. For example, while *OFF*=0 the execution of the rectangular node a^0_3 is repeated. If *OFF*=1 the module Z_0 is terminated. Rectangular nodes contain micro-operations, which affect actuators of the embedded controller forcing the required operations. For example, the operation *move left* forces some movable element(s) to be in motion to the left.

One module Z_a might activate another module Z_b in such a way that: a) the module Z_a has to be suspended; b) the module Z_b has to be executed; c) as soon as the module Z_b is terminated, the control has to be returned back to the module Z_a, i.e. the module Z_a has to continue its execution starting from a node following the node which called Z_b. For example, the node a^0_2 of the module Z_0 (a=0) activates the module Z_6 (b=6). After the Z_6 is terminated, the control has to be returned back to Z_0 and the node a^0_3 has to be activated.

If two or more modules are activated in the same node they have to be executed in parallel. For example, the modules Z_1 and Z_2 have to be activated in parallel from the module Z_0. If two or more modules are called in parallel from the module Z_a, the module Z_a is allowed to continue its execution if and only if all called parallel modules have been terminated. In other words, if any of parallel modules is still functioning, the module Z_a has to remain suspended.

Fig. 2 presents the proposed model of PHFSM which is based on the known model of HFSM [3] and is capable to execute PHGSs (such that is shown in Fig. 1).

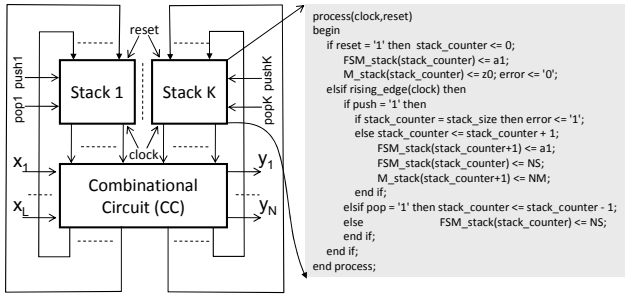

Figure 2. The model of Parallel Hierarchical Finite State Machine

There are K stacks in Fig. 2 connected to a common combinational circuit (CC). Each stack has associated inputs *push* and *pop* and all the stacks have shared *reset* and *clock* inputs. The number K is equal to the maximum number of modules running in parallel. Each stack can be constructed using a common template for memory of HFSM considered in detail in [6]. The right-hand side of Fig. 2 depicts such template coded in VHDL.

Let us consider a graph Γ, which has the same number of nodes *N* that the number of modules in PHGS. Each node i (i=0,…,N) of the graph is associated with the module i, and the node *a* is connected by a directed edge with the node *b* if and only if the module Z_b has to be activated from the module Z_a.

Fig. 3 depicts the graph Γ constructed for the PHGS in Fig. 1. The maximum number of parallel branches in the graph Γ gives the maximum number of modules running in parallel. Note, that for an arbitrary activation of different modules, discovering the maximum number is a very complicated task, which is unsolvable for some cases. Thus, let us introduce the following constraints:

- The activation of the parent module in branches running in parallel is prohibited;
- Parallel calls in any recursive module are not allowed.

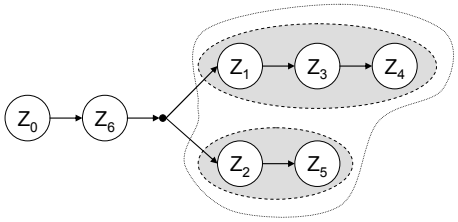

Figure 3. The graph Γ for the modules in Fig. 1

Taking into account the considered above constraints the maximum number of modules running in parallel can be

calculated through examination of the graph Γ. Finally, it gives the value of K and for our example K=2.

The next step has to associate modules with the stacks and to determine the size of stack words. The following rules are proposed:

- Examining a parallel set with the maximum number K of modules running in parallel and building K stacks. The size of words for each $stack_k$ (k=1,…,K) is assigned either $int\log_2 M_k$ (M_k is the number of states for the module k) for binary state encoding [2] or M_k for one-hot state encoding [2];
- Examining the remaining parallel sets and constructing their memories from the stacks that have already been constructed in such a way that:

a) Any previously constructed stack can be entirely reused;

b) Any new stack can be build from more than one constructed stacks;

c) Any new stack can be build from a previously constructed stack through increasing the size of the words. In this case the size of words will also be increased for the constructed stack.

After applying the considered above rules all stacks will be constructed. To minimize their size it is necessary to solve an optimization task for the second rule, which can be converted to the matrix covering problem [7]. The result of the covering can undoubtedly be found taking into account the possibility of increasing the size of stack words.

Section IV shows how to use the model in Fig 2 for synthesis of PHFSM that implements the algorithm in Fig.1 for a simple embedded controller.

III. An Example

Figure 4 depicts two interacting objects of a self-controlled transport section: a robot on the left hand side; and a container on the right-hand side. Both the robot and the container can move from left to right and vice versa. A similar example was considered in [5], but comparing with [5] it is extended by additional macro-operations $Z_3,\ldots,Z_6$ making it possible a number of relatively complicated processes to be executed. Functionalities of both the robot and the container have to be controlled in parallel. The relevant algorithms (Z_1 for the robot and Z_2 for the container) are shown in Fig. 1.

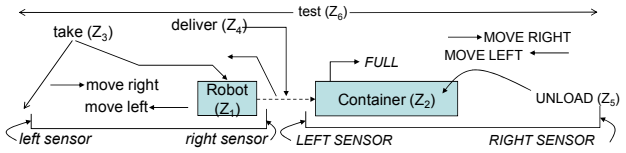

Figure 4. An example of a self-controlled transport section

The following macro-operations have to be executed:

- Z_1 – controlling functionality of the robot;
- Z_2 – controlling functionality of the container;
- Z_3 – controlling the robot when it takes something (on the left-hand side);
- Z_4 – controlling the robot when it delivers something to the container;
- Z_5 – controlling unloading of the container;
- Z_6 – verifying functionality of the entire system.

The following micro-operations have to be executed:

- move right/MOVE RIGHT – the robot/the container has to be moved right;
- move left/MOVE LEFT – the robot/the container has to be moved left;
- wait/WAIT – the robot/the container has to wait;

The following sensors (logical conditions) are used:

- left sensor/LEFT SENSOR – left sensor for the robot/ container transport section;
- right sensor/RIGHT SENSOR – right sensor for the robot/ container transport section;
- FULL – the container is full.

In accordance with the algorithm in Fig. 1, the robot is controlled by two actuators *move left* and *move right*, which force the respective motions. The track, where the robot is moving, is bounded by the sensors *left sensor* and *right sensor* in such a way that if *left sensor*=1/*right sensor* =1, the robot is at the left/right edge.

The container has exactly the same behavior, except capital letters are used instead of lower-case letters in the description. The transfer is also controlled by three macro-operations Z_3-Z_5 in such a way that Z_3 forces the robot to take something at the left-hand side of the transfer line; Z_4 delivers the object carried by the robot to the container; and Z_5 unloads the container at the right-hand side. Macro-operation Z_6 tests the entire system before initiating the working stage. The algorithm in Fig. 1 assumes that several iterations are needed for the robot to load the container.

Taking into account all the considered above details we can conclude that Fig. 1 describes the functionality of a real-world system. To simplify the problem we will not give details of the modules Z_3-Z_6 (in particular, in the module Z_3 all micro-operations/logical conditions are abstract and that is why they were not characterized above). Any of the modules Z_3-Z_6 describes functionality of a *simple sequential* FSM, i.e. they are not hierarchical.

Let us apply the introduced in the previous section model to the system shown in Fig. 4. The next section presents a method of synthesis of the relevant PHFSM.

IV. Synthesis of Parallel Hierarchical Finite State Machines

The problem of synthesis is formulated in the same way as in [3], i.e. for a given control algorithm Λ, described by a set of PHGSs, construct the PHFSM that implements Λ.

The synthesis includes the following steps:

- Marking the given PHGSs with labels corresponding to the PHFSM states;
- Describing the required functionality of the CC (see Fig. 2) with the aid of reusable hardware description language (HDL) code, which we will call HDL template for the CC;
- Describing the stacks with the aid of HDL templates discussed in [6];
- Providing predefined (see Fig. 2) connections between the stacks and the CC;
- Synthesis of PHFSM using any available computer-aided design system for the chosen HDL.

In order to mark given PHGSs, it is necessary to do the following (see Fig. 1):

- The first label (let us call it a^i_1) is assigned to the node Begin of all modules Z_i (i=1,…,N);
- The last label (let us call it a^i_{Mi}) is assigned to the node End of all modules Z_i (i=1,…,N) and M_i is the number of states in the module i (i=1,…,N);
- The labels $a^i_2,\ldots,a^i_{Mi-1}$ are assigned to unmarked rectangular nodes in all modules Z_i (i=1,…,N).

Now the labels are considered to be PHFSM states. The states for modules associated with the same stack are not distinguished by their superscripts. In other words, as soon as we code them in HDL we have to use the same name like a_2 for the states a_2 with different superscripts such as a^0_2, a^1_2, a^2_2, etc. This can be done thanks to the name of module, which has also to be provided. Thus, the pair ia_m (i.e. the module name i and the state name a_m) allows knowing both the module and the state and additional information about superscript is not required. This permits, in particular, using the same codes for states of different modules with the same subscript.

The proposed template for the CC has the following VHDL code (any other HDL can be used in a similar way):

```
process («listing all inputs of the CC»)
begin
-- all push and pop signals are assigned '0';
  case M_stack1(stack_counter1) is
    when Z0 =>  -- the code of this module is
                -- given for Z0 in Fig. 1
      case FSM_stack1(stack_counter1) is
        when a1 => -- assigning outputs and
              -- executing state transitions
          NS1<=a2;    -- NS is the next state
        when a2 =>
          -- hierarchical calls if required
          NS1<=a3;  push1 <= '1';
           NM1 <= Z6; -- NM is the next module
      -- here and below avoiding iterative
      -- module invocations with the aid of
      -- methods [6, pp. 200-201]
        when a3 =>
          -- parallel calls if required
           if OFF='1' then NS1 <= a4;
           else NS1 <= a3;
           end if;
           push1 <= '1'; push2 <= '1';
           NM1 <= Z1; NM2 <= Z2;
        when a4 => NS1 <= a4;
                -- terminating node
        when others => null;
      end case;
      when Z1 => -- the code of this module is
      -- given just for hierarchical
      -- calls/returns in Z1 (Fig. 1)
        case FSM_stack1(stack_counter1) is
          when a3 =>
            if right_sensor='1' then
              NS1 <= a5;
            else NS1 <= a4;
            end if;
            push1 <= '1'; NM1 <= Z3;
          -- . .. . . . . . . . . . .
          when a6 => NS1 <= a7;
             push1 <= '1'; NM1 <= Z4;
          when a7 => NS1 <= a7;
            if ((stack_counter1 > 0) and
           (FSM_stack2(stack_counter2) = a6))
            then pop1 <= '1';
             else pop1 <= '0';
             end if;
           when others => null;
        end case;
        when Z3 =>  -- repeat for all modules
           -- associated with the first stack
        when others => null;
      end case;
      when others => null;
    end case;

    case M_stack2(stack_counter2) is
      when Z2 => -- repeat for all modules
        -- associated with the second stack
      when others => null;
    end case;
    when others => null;
  end case;
   -- repeat for all modules associated
   -- with the remaining stacks
end process;
```

In the template above output signals are skipped for the simplicity. Assuming that we deal with Moore FSM model [1] the outputs can be generated in the relevant states, for example, in the state a^1_2: `move_left <= '1'`. The complete synthesizable VHDL code of the CC for all modules shown in Fig. 1 is available online at [8]. Note that the code [8] contains all necessary additional lines allowing to avoid repeated module invocations in accordance with the suggestions [6, pp. 200-201].

A simplified template for stacks is given in Fig. 2. The complete synthesizable VHDL code of stacks for PHGS in Fig. 1 is available online at [8]. Each $stack_k$ contains two parts (memories) that are a stack of modules (M_stack_k) and a stack of states (FSM_stack_k). They are addressed by a shared $stack\_counter_k$ incremented by the signal $push_k$ and decremented by the signal pop_k. The stack M_stack_k keeps the

codes of modules and the stack FSM_stack_k stores the codes of states. The signals NM_k (NS_k) supply the code of the next module (the code of the next state) formed by the template for the CC. All the details about stack memories for HFSM can be found in [6]. The only difference for PHFSM is a necessity for multiple stacks. However, each individual stack is exactly the same as in [6] and thus, it will not be described once again.

There are totally 5 different types of state transitions:

1) Simple sequential state transitions between states within the same module. They are provided in the same way as in an ordinary FSM, for example the transition from a^1_5 will be done like the following:

```
when a5 =>
     if ((FULL='0') and (L_SENSOR='1')) then
       NS1 <= a6;
     else
       NS1 <= a5;
     end if;
```

2) Simple (non parallel) hierarchical transitions, for example, a hierarchical transition from the state a^0_2 (the signal `return_flag` is needed to avoid the second invocation of the same module Z_6 during the return step, which is done in accordance with the method [6, pp. 200-201]):

```
  when a2 =>
    NS1 <= a3;
    if return_flag1 = '0' then
      push1 <= '1'; NM1 <= Z6;
    else
      push1 <= '0';
    end if;
```

3) Simple (non parallel) hierarchical returns, for example, a hierarchical return from the state a^3_4:

```
when a4 =>
  NS1 <= a4;
  if (stack_counter1 > 0) then
    pop1 <= '1';
  else
    pop1 <= '0';
  end if;
```

4) Parallel hierarchical calls, for example, a parallel hierarchical call from the state a^0_3:

```
when a3 =>
  if OFF='1' then NS1 <= a4;
  else NS1 <= a3;
  end if;
  if return_flag1 = '0' then
    push1 <= '1'; push2 <= '1';
    NM1 <= Z1; NM2 <= Z2;
  else push1 <= '0'; push2 <= '0';
  end if;
```

5) Hierarchical returns from parallel branches, for example, the return from the state a^2_6:

```
when a6 =>
  NS2 <= a6;
  if ( (stack_counter2 > 0) and
     (FSM_stack1(stack_counter1) = a7)) then
    pop2 <= '1';
  else pop2 <= '0';
  end if;
```

Fig. 5 demonstrates the functionality of PHFSM for PHGS in Fig. 1 for state transitions indicated by different directed curves. Note, that one-hot state encoding is considered just for simplicity. Obviously binary state encoding can also be used.

Now the predefined connections have to be provided between the CC and the stacks in accordance with Fig. 2. Finalizing the synthesis can be done in any appropriate design environment. In our particular case it was done in ISE 9.2 of Xilinx [9].

V. EXPERIMENTS

The primary goal of the experiments was to prove on arbitrarily selected working examples that the model and the method presented in the paper are correct. Different types of parallel control algorithms with up to 5 parallel branches and up to 14 hierarchical calls were analyzed. Various PHFSM implementing these algorithms on the basis of recent commercially available low-cost FPGAs were synthesized, implemented, and tested.

We have used the stand-alone boards DETIUA-S3 [10] and TE-XC2Se [11]. Note that the first board supports both wired (USB) and wireless (Bluetooth) interfaces for configuring the onboard FPGA and interacting with the implemented circuits. This permits to construct remotely modifiable parallel control systems, which are very useful for numerous experiments in such areas as embedded controllers, industrial electronic devices, etc.

The implemented PHFSMs require reasonable FPGA resources. For example, the complete circuit functioning in accordance with the specification in Fig. 1 occupies just 8% of slices of a very cheap xc2s400e FPGA [9] and other resources, such as embedded memory blocks, have not been used. Thus, very complicated PHFSM can be implemented on the basis of low-cost FPGAs. The results of experiments have shown that the proposed model and methods are indeed very practical and enable the designers to implement real-world hierarchical and/or parallel algorithms. This, in particular, permits to modify the conclusion given in [5], which says that "although hierarchical and parallel algorithms can be implemented within the same system, using hierarchical parallel algorithms is very resource consuming and many constraints have to be taken into account. In many cases, autonomous HFSMs working in parallel are better". This paper presents new results making it possible to conclude that hierarchical parallel algorithms are, in fact, not so resource consuming and those constraints, that have to be taken into account, are not so important for many of real-world problems.

The complete project for Xilinx ISE 9.2 [9] and stand-alone prototyping board TE-XC2Se (that is ready for evaluation and test) is available online at [8].

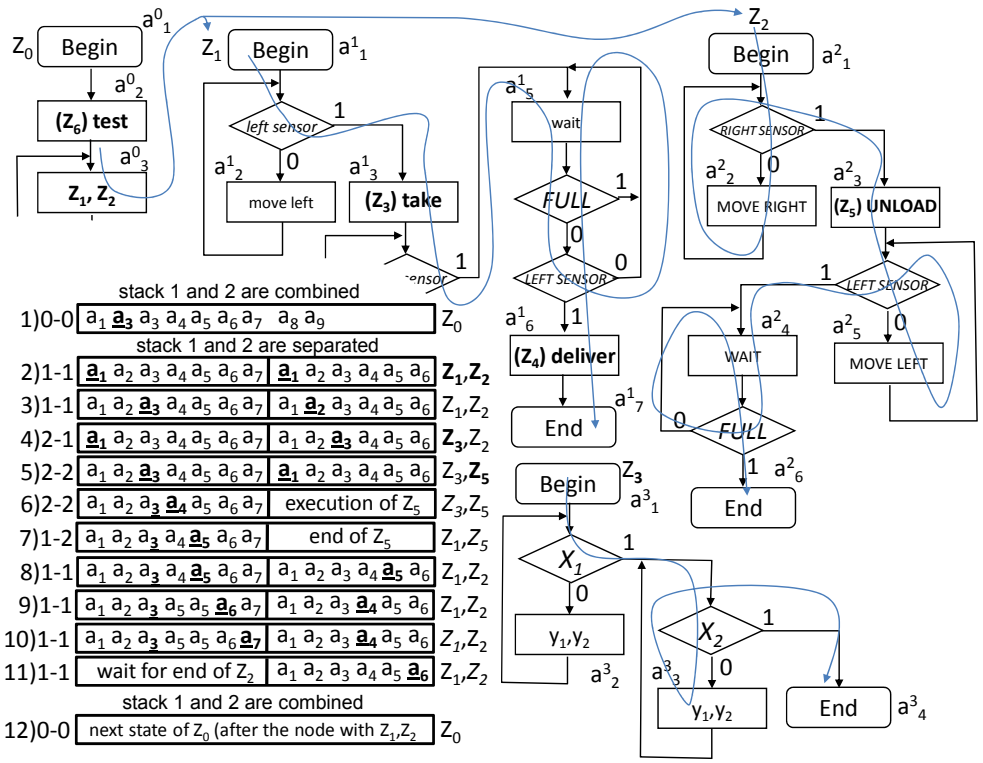

Figure 5. An example demonstrating different types of state transitions

VI. Conclusion

The paper presents a novel model and method of synthesis of parallel hierarchical finite state machines. The latter allow both hierarchical and parallel calls of algorithmic modules to be implemented. Any module is considered to be either a simple finite state machine (FSM), a hierarchical FSM or a parallel (hierarchical) FSM. Although there are some constraints, they are not so important for the majority of real-world problems. The primary contribution is that implementation of hierarchical parallel algorithms on the basis of the proposed model is very practical and consumes very reasonable hardware resources.

References

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

[2] T.Villa, T.Kam, R.K.Brayton, A.Sangiovanni-Vincentelli, Synthesis of Finite State Machines: Logic Optimization, Kluwer Academic Publishers, 1997.

[3] V. Sklyarov, “Hierarchical Finite-State Machines and their use for digital control”, IEEE Transactions on VLSI Systems, 1999, vol. 7, no. 2, pp. 222-228.

[4] A. Zakrevskij, Parallel Algorithms of Logical Control, Minsk, Academy of Science, 1999.

[5] V. Sklyarov, I. Skliarova, "Hierarchical specification and design of control systems in robotics", Proceedings of the 3rd Int. Conf. on Autonomous Robots and Agents - ICARA'2006, Palmerston North, New Zealand, December 2006, pp. 623-628.

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

[7] I. Skliarova, Reconfigurable Architectures for Problems of Combinatorial Optimization, Ph.D. Thesis, University of Aveiro, Portugal, 2004.

[8] Synthesizable VHDL code for PHFSM, Online: http://www.ieeta.pt/~skl/Research/Projects/ISE_Projects/ParallelHFSM.rar.

[9] Xilinx, Inc., products and services, Online: http://www.xilinx.com.

[10] M. Almeida, B. Pimentel, V. Sklyarov, I. Skliarova, "Design tools for rapid prototyping of embedded controllers", Proceedings of the 3rd International Conference on Autonomous Robots and Agents - ICARA'2006, Palmerston North, New Zealand, December 2006, pp. 683-688.

[11] Spartan-IIE Development Platform, Online: http://www.trenz-electronic.de.