

Synthesis and Implementation of Hierarchical Finite State Machines with Implicit Modules

Valery Sklyarov
 DETI/IEETA,
 University of Aveiro,
 Aveiro, Portugal
skl@ua.pt

Iouliia Skliarova
 DETI/IEETA,
 University of Aveiro,
 Aveiro, Portugal
iouliia@ua.pt

Dmitri Mihhailov
 Computer Department,
 TUT,
 Tallinn, Estonia
d.mihhailov@ttu.ee

Alexander Sudnitson
 Computer Department,
 TUT,
 Tallinn, Estonia
alsu@cc.ttu.ee

Abstract—The paper describes a hierarchical finite state machine (HFSM) with implicit modules, which inherits capabilities of existing models (in particular, provides support for modularity, hierarchy, and recursion); requires a very simple stack memory; and permits optimization methods developed for conventional FSMs to be reused. The HFSM has been tested in several practical applications briefly characterized in the paper. It is shown that the same hardware can implement different algorithms through the proposed reconfiguration technique. The results of experiments, reported in the paper, clearly demonstrate advantages of the proposed model.

Keywords—finite state machines; circuit synthesis; hierarchical graph-schemes; reconfiguration technique

I. INTRODUCTION

Finite state machines (FSM) are probably the most widely used components in digital circuits and systems. That is why almost all the available automatic design tools that are included in industrial CAD systems allow FSMs to be synthesized from their formal specifications [1], such as state diagrams, state transition tables, hardware description language (HDL) descriptions, etc. All these specifications are appropriate for the design of relatively simple circuits. For more complex circuits it is very important to provide support for enhanced features, namely hierarchical description of FSM functionality at different levels of abstraction. Such more advanced FSM have been studied in a number of publications. In [2] the model of hierarchical FSM (HFSM) based on the use of stack memory and methods of synthesis of HFSM from specification in form of hierarchical graph-schemes (HGS) were proposed. HGS can be seen as an extension of graph-schemes [3]. A preliminary study of HFSMs with multiple concurrency models (namely dataflow, synchronous reactive systems, and discrete-event systems) was done in [4], where also some possible enhancements were indicated. Practical applications of HFSMs were analyzed in numerous publications. For example, the results of [2], further improved in [5], were considered to be convenient for the design of hardware and software for complex data processing operations in [6], for models of computations described in [7], for control system in Medusa instrument [8], etc. They can be used at different levels, for example for local control in [9] and for implementation of relatively complex embedded systems [10]. Statecharts [11], which can be seen as another type of

specification applicable to HFSMs, were adapted for object-oriented programming and used as a part of unified modeling language (UML). Many papers are dedicated to synthesis of HFSMs from scenario-based notations such as UML (e.g. [12,13]) for software development. A number of practical applications require software components to be implemented in hardware circuits. In [14] HFSMs are used for reconfigurable SoC design where hierarchy is important at system level. Thus, methods and tools for synthesis of such circuits and systems are important and mechanisms available for software development (e.g. recursion) need to be provided in hardware.

Note that the model of HFSM [2,5] permits also recursive calls to be implemented in hardware. The work [15] extends the results of previous proposals and suggests a data-oriented approach. Other known techniques are presented in survey [16], which shows that the model [2,5] permits any recursive function to be implemented and it leads to such circuits that occupy medium resources and are comparable to other competitive techniques. Besides, the model [2,5] supports parallel execution of macro operations (sub-algorithms) [17]. However, this model does not permit to apply the majority of optimization methods developed for conventional FSMs and requires resource consuming stacks. Practicability of this model can be extended if we are able to apply reconfiguration technique that allows the same hardware to be used for implementing different HGSs.

This paper describes a model of HFSM that provides support for all features indicated above (namely: modularity, hierarchy, recursion, and parallelism); requires just a very simple stack memory; and permits optimization methods developed for conventional FSMs to be entirely reused. It is also shown how reconfiguration technique can be applied.

The remainder of the paper is organized in five sections. Section II describes the known technique for specification and synthesis of HFSMs. Section III describes HFSMs with implicit modules and gives an example. Section IV suggests a reconfiguration technique. Section V describes implementations of HFSMs and experiments. The conclusion is in Section VI.

II. SPECIFICATION AND SYNTHESIS OF HFSM

Basically, hierarchy is useful for two kinds of applications [18] that are:

- Autonomous sequential modules that can be considered as components of more complicated digital systems. For

example, the paper [19] examines an FSM that reads a sequence of bits and detects in the sequence two or more successive ones. The relevant FSM can be designed and used as a component of more advanced FSMs.

- Control circuits. For example, in [20] an HFSM was used for variable instruction set processor. It permits to optimize the processor by means of reusing similar fragments in micro-programs for different operations.

We will consider both kinds of applications and describe the desired functionality using HGSs. An HGS [2,5,18] is a directed connected graph containing rectangular, rhomboidal and triangular nodes. Each HGS has one entry point, which is a rectangular node named *Begin*, and one exit point, which is a rectangular node named *End*. Other rectangular nodes contain either *micro instructions* or *macro instructions*, or both. We will allow also *micro instructions* to be assigned to the nodes *Begin* and *End* if required. Any *micro instruction* Y_j includes a subset of *micro operations* from the set $Y=\{y_1, \dots, y_N\}$. A *micro operation* is an output binary signal. Any *macro instruction* incorporates a subset of *macro operations* from the set $Z=\{z_1, \dots, z_Q\}$. Each *macro operation* is described by another HGS of a lower level considered as a *module*. If a macro instruction includes more than one macro operation then all these macro operations have to be executed in parallel. Each rhomboidal node contains one element from the set X , where $X=\{x_1, \dots, x_L\}$ is the set of *logic conditions*. A *logic condition* is an input binary signal, which communicates the result of a test. Directed lines (arcs) connect the inputs and outputs of the nodes in the same manner as for a graph-scheme [3]. Each triangular node contains an expression which permits an output of this node to be selected. When the control flow passes a triangular node, exactly one output is activated enabling the control flow to proceed. The output of a rectangular node k containing more than one element z_i, z_j, \dots from the set Z is called a merging point. Control flow passes the merging point if and only if all the elements z_i, z_j, \dots have been completed. All other details can be found in [2,5,18].

Using HGSs enables us to develop any complex algorithm step by step, concentrating efforts at each stage on a specified level of abstraction [5]. Each separate HGS (module) is usually very simple, and can be checked and debugged independently.

Fig. 1 describes an algorithm by HGSs. There are totally 5 different modules. An execution starts from the node *Begin* of the main HGS (module) z_0 and terminates at the node *End* of z_0 . The HGS z_0 can call modules z_1 , z_2 and z_3 . The module z_2 can activate itself recursively and call z_4 .

An HGS can be converted to an HFSM using the model and methods [2,5] and then formally described in a hardware description language using the HDL templates proposed in [21]. The resulting (customized) HDL code is synthesizable in commercially available CAD systems (e.g. Xilinx ISE).

The model [2,5] possesses the following features. There are two stack memories that keep words of $\lceil \log_2 Q \rceil$ bits for modules and $\lceil \log_2 R \rceil$ bits for states, where Q is the number of modules and R is the maximum number of states in a module. States in different modules can be assigned the same

labels and thus the same codes. Therefore, a module and a state of the active module are selected explicitly by the top registers of both stacks. We will call the known model *HFSM with explicit modules*. Additional details about this model can be found in [21].

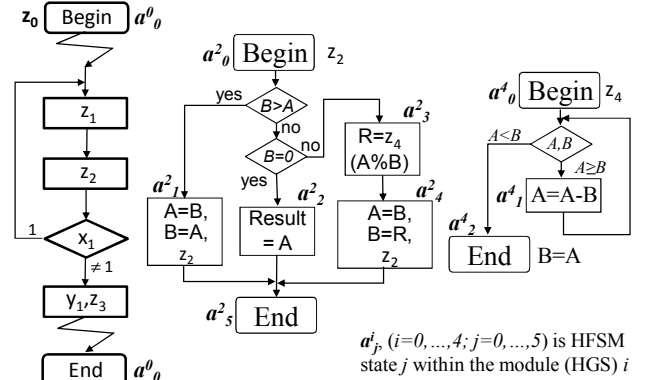


Figure 1. Example of HGSs

III. HFSM WITH IMPLICIT MODULES

HFSM with *implicit modules* [18] (see Fig. 2), behaves like an ordinary FSM and a single stack of states is used just for returns from the called modules.

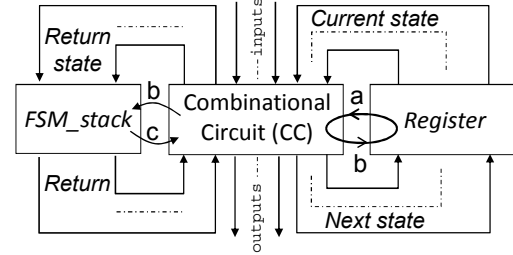


Figure 2. HFSM with implicit modules

There are three types of state transitions in this model (see Fig. 2) that are:

- Ordinary transitions (a), which are executed between states that belong to the same module;
- Hierarchical calls (b) of other (called) modules from the currently executing (calling) modules;
- Hierarchical returns (c) from the previously called modules to the calling modules.

The first two types of transitions are executed through a *Register* much like this is done in conventional FSMs (see (a) and (b) in Fig. 2). In the first type, states in the *Register* are changed from the current a_c to the next a_n ($a_c \rightarrow a_n$). In the second type the current state a_c in the *Register* is changed to the first state a_f of the called module HGS_{called} . The state a_c of the calling module $HGS_{calling}$ is saved in the stack because it is needed later to execute the correct state transition in the $HGS_{calling}$ during the relevant hierarchical return. Thus, just return states have to be saved in the stack.

The number of such states is, as a rule, relatively small. Therefore the stack requires small resources.

The states are assigned to all rectangular nodes of HGSs in such a way that:

- The state a_0 is assigned to the node *Begin* of the main HGS z_0 . Note that a_0 can also be assigned to the node *End* of z_0 if z_0 has to be executed cyclically.
- The states a_1, a_2, \dots, a_{M-1} are assigned to unmarked rectangular nodes in all HGSs.
- The states cannot be repeated (except the state a_0 in the HGS z_0).
- All rectangular nodes have to be labeled with states.

After labeling the HGSs the considered above transitions have to be implemented.

The described model supports all features of the model [2,5,21] and needs a single simple stack. Since the codes of all states are unique and the modules are hidden (implicit), all known optimization methods that are used for conventional FSMs can be applied directly.

Experimental results (see section V) demonstrate that the HFMSMs with implicit modules are faster and less resource consuming. There is also an opportunity to apply known optimization methods that have been developed for conventional state machines. The HFMSM model with explicit modules [2,5,21] is not so well suited for such optimization, mainly because states in different modules can be assigned the same codes. The modules in HFMSMs in Fig. 2 are implicit and cannot be updated and refined easily. The next section shows that reconfiguration technique can be applied for the both models but for the model in Fig. 2 the complete HFMSM has to be reconfigured.

Let us slightly modify the model shown in Fig. 2 in such a way that codes in the register are decomposed in two parts: codes of modules (part 1) and codes of states (part 2). In this case similarly to [2,5,21] the codes of states can be the same in different modules. Thus, some constraints are applied in case of the use of optimization methods developed for conventional state machines, but the resource consumption is nearly the same as for the model in Fig. 2 (see Fig. 3). Besides, reconfiguration technique can easily be applied to individual modules (see the next section).

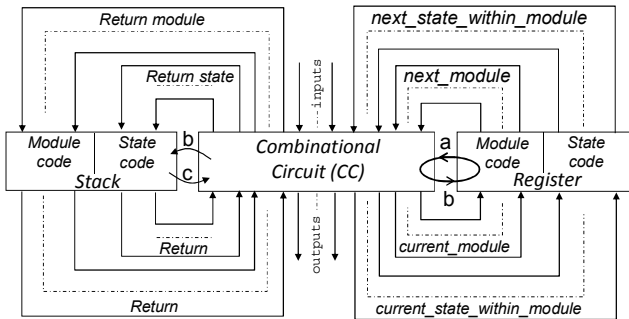


Figure 3. Different encoding of modules and states for HFMSM in Fig. 2

Let us consider an example shown in Fig. 1 for the modules z_2 and z_4 which permit to find the greatest common divisor (GCD) for two unsigned integers using recursive

algorithm. All necessary details for the algorithm are given in [18]. Since for the model in Fig. 3 states in different modules can be assigned the same codes we will not consider superscripts for states in Fig. 1. The combinational circuit can be described by the following VHDL code:

```

push <= '0'; pop <= '0'; N_M <= RgM;
case RgM is -- for such simple circuit the signal Return_M can be
when z2 => -- removed because the return is always done to z2
  case RgS is
  when a0 => A <= Data_A; B <= Data_B;
    if Data_B > Data_A then N_S <= a1;
    elsif Data_B = 0 then N_S <= a2;
    else N_S <= a3;
    end if;
  when a1 => push <= '1'; Data_A <= B; Data_B <= A;
    N_S <= a0; N_M <= z2; Return_S <= a5;
    Return_M <= z2;
  when a2 => N_S <= a5; result <= A;
  when a3 => push <= '1'; N_S <= a0; N_M <= z4;
    Return_S <= a4; Return_M <= z2;
  when a4 => push <= '1'; Data_B <= R; Data_A <= B;
    N_S <= a0; N_M <= z2; Return_S <= a5; Return_M <= z2;
  when a5 => N_S <= a5;
    if stack_pointer > 0 then pop <= '1';
    else pop <= '0';
    end if;
  when others => null;
  end case;
when z4 =>
  case RgS is
  when a0 =>
    if A >= B then N_S <= a1;
    else N_S <= a2;
    end if;
  when a1 => A <= A - B; R <= A - B; N_S <= a0;
  when a2 => N_S <= a2;
    if stack_pointer > 0 then pop <= '1';
    else pop <= '0';
    end if;
  when others => null;
  end case;
when others => null;
end case;

```

Here: RgM/RgS is the part of the same *Register* (see Fig. 3) that stores module/state codes; $push/pop$ are signals that increment/decrement stack pointer ($stack_pointer$); $Data_A$, $Data_B$ are input integers for which GCD has to be found; A , B are temporary signals to which $Data_A$, $Data_B$ are copied at the beginning; N_S/N_M is the next state/the next module in the HFMSM; the signals $Return_S/Return_M$ keep states that have to be stored in the stack (all necessary details for such signals can be found in [18]); R is the result of the module z_4 . Additional explanations can be found in [18] where an example is considered for the model shown in Fig. 2 ($Return_M$ is implemented similarly to $Return_S$).

We assume in Fig. 1 that the module z_0 receives two integers $Data_A$ and $Data_B$ through an interface provided by the module z_1 . The considered above module z_2 calculates GCD of $Data_A$ and $Data_B$ and GCD is considered as a condition x_1 . If $x_1 = \text{GCD} = 1$ then a new pair of integers is received by z_1 , otherwise the GCD is processed in the module z_3 . The module z_4 is used to calculate $A\%B$ (the remainder that results from the division of A by B). Here, $a^0_0, \dots, a^2_0, \dots, a^5_0, \dots, a^t_0, \dots, a^t_2$ are HFMSM states that are

introduced using the rules [5]. Further these states are used in the modules z_2 and z_4 without superscripts (see in VHDL code above the labels a_0, a_1, \dots, a_5 for the module z_2 and the labels a_0, a_1, a_2 for the module z_4). There are three module calls in z_2 in the states a_1, a_3 and a_4 . Returns from a_1, a_3 and a_4 can be to either a_4 in z_2 (when z_4 is terminated) or a_5 in z_2 (when z_2 is terminated). The states a_4 and a_5 can be uniquely distinguished by 1-bit binary vectors (e.g. 0 for a_4 and 1 for a_5) attaching the relevant *encoder* and *decoder* [18]. The modules z_2 and z_4 for our example can also be uniquely distinguished by 1-bit binary vectors. These vectors are saved in a 2-bit register of the stack (see Fig. 3) and each vector indirectly points out which state transition in which module has to be done on a hierarchical return. For example, if z_4 is terminated, the transition to the state a_4 in the module z_2 has to be done. Note that module calls (such as z_4) might appear in different states and in different modules, and, thus, the returns have to be provided to different states in different modules from the same called module. That is why the returned state (such as a_4 or a_5) and the returned module have to be chosen in calling modules (but not in called modules). The called modules might change conditions that influence transitions from the states of calling modules. Such changes are taken into account with the aid of the methods proposed in [18,21]. VHDL codes for the single stack and for the *Register* are the same as in [18] with the only difference that a pair $\langle \text{module_code}, \text{state_code} \rangle$ has to be used instead of just *state_code* in [18], where the *state_code* is the code of state and *module_code* is the code of module. It means that in each state transition, hierarchical call and hierarchical return both *module_code* and *state_code* have to be taken into account. The stack is needed just to know which state and module have to be the target of a transition when a called module is terminated. Note that the return from a called module and the transition from the relevant state of the calling module are executed within the same clock cycle. In any module all the necessary state transitions are realized through the *Register* (see Fig. 2 and Fig. 3), much like the way that it is done in a conventional FSM. Suppose that a new module z_n has to be called in a state a_c of the module z_c . When the called module z_n is terminated, the stack pointer is decremented and the selected top register of the stack points to a state a_R in z_c , where a_R is the next state in z_c after termination of z_n . There exist two modes of returns. In the first mode there are no conditional transitions from the state a_c . Thus, a_R is the unconditional next state from a_c in the calling module. This mode is considered for our example. In the second mode there are conditional transitions from the state a_c and they can be influenced by the called module z_n . In this case the method based on the use of a special return flag (described in [21] with all necessary details) can be applied directly. It is important to note that the transition from this state (a_c) will be done during the same clock cycle with hierarchical return and no delay is introduced [21]. This is achieved through the technique described below. Let us consider a couple “ $a_c z_n$ ”, where a_c is the state of a calling module z_c from which the module z_n is called (thus, z_n is a called module). All pairs $\langle a_c, z_c \rangle$ and $\langle a_c, z_n \rangle$ are unique in a sense that they cannot be

repeated with the same $\langle a_c, z_c \rangle$ and z_n . When z_n is called, the following steps are executed: 1) the code of $\langle a_c, z_c \rangle$ is saved in the top register of the stack; 2) the codes of z_n and the first state of z_n are saved in the *Register*; 3) the stack pointer is incremented. Thus, the stack keeps track of returns beginning with the currently active module. When any called module is terminated, the following steps are performed: 1) the transition from the state a_c within the module z_c , both indicated by *stack_pointer-1*, is executed; 2) the stack pointer is decremented. Finally, no extra delay is introduced in the HFSM compared to a conventional FSM. Additional details can be found in [18,21].

IV. RECONFIGURATION TECHNIQUE

Reconfiguration enables the same HFSM to be used to implement different HGSs. To provide such technique we will apply methods described in [22], where CC of FSM is built from RAM blocks. Fig. 4 depicts the proposed dynamically reconfigurable CC for the considered HFSM.

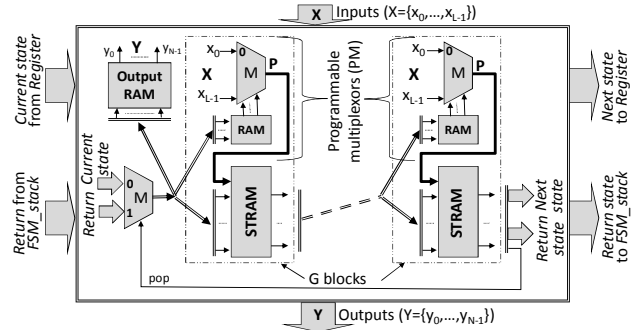


Figure 4. A reconfigurable combinational circuit for HFSM

There are G RAM-based blocks in the circuit and each block is composed of a programmable multiplexer (PM) and a state transition RAM (STRAM). PM enables L input variables from the set X to be replaced with smaller number of variables from the set $P = \{p_0, \dots, p_{K-1}\}$. As an example, Fig. 5a shows how two input variables x_1 and x_3 can be replaced in a state a_m with variables p_0 and p_1 (in this case $K=2, L=4$). Fig. 5c demonstrates programming of STRAM for some arbitrary 3-bit codes of HFSM states for a fragment of HGS shown in Fig. 5b. RAM in PM converts state codes to such codes that enable necessary inputs to be selected and replaced with variables from the set P .

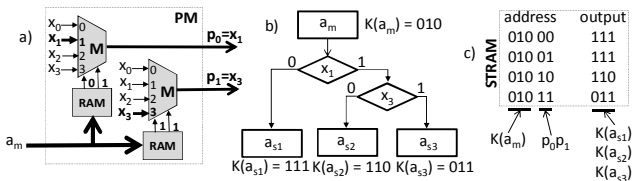


Figure 5. An example

Reconfiguration is achieved with the aid of an additional controller described in [22]. As we can see from Fig. 4 the complete HFSM circuit can be customized to implement a

new functionality. There are some constraints applied to the circuits that are the same as in [22].

Similar technique can be offered for individual modules and, in this case, two additional advantages can be attained:

- Time required for reconfiguration is reduced;
- Reconfiguration of a module can be done while other modules are active.

Fig. 6 presents the details. All modules have to be explicitly coded. For each module a dedicated CC is used. Codes of states and outputs are taken just from an active module and outputs of passive modules are set to zeros.

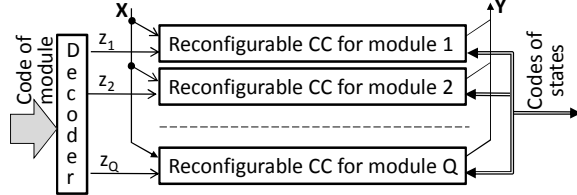


Figure 6. Reconfiguration of individual modules

The circuit in Fig. 6 is more flexible and more resource consuming. It can be used for HFSM with explicit modules [5] and for HFSM shown in Fig. 3.

V. IMPLEMENTATIONS AND EXPERIMENTS

HFSM models and methods of synthesis were examined for the following applications:

- An embedded system for garage control [10].
- A processor with variable instruction set [20].
- Iterative and recursive modular sorting algorithms [21].
- Priority buffers (queues) for the embedded system [10].

All indicated applications were implemented and tested in FPGA. In addition, the methodology was used in educational process and by numerous Ph.D. students working on computationally intensive algorithms.

The synthesis and implementation of the circuits from specification in VHDL were done in Xilinx ISE for FPGA Spartan3E-1200E-FG320 of Xilinx available on prototyping board NEXYS-2 of Digilent and for other FPGAs of Spartan3E family indicated in [10,20].

Table I presents the results of synthesis for HFSM with explicit and implicit modules for different sorting algorithms A_1, \dots, A_5 , where S is the number of slices, B is the number of block RAMs (block RAMs were used to construct stacks and other circuits required for implemented algorithms) and F_{max} is the maximum attainable clock frequency.

TABLE I. THE RESULTS OF SYNTHESIS FOR SORTING ALGORITHMS

Algorithm	HFSM (explicit)			HFSM (implicit)		
	S	F_{max}	B	S	F_{max}	B
A_1	2415	100	13	1175	107	13
A_2	4407	100	27	1965	103	27
A_3	2609	77	20	1146	65	20
A_4	2812	75	20	1312	83	20
A_5	3343	73	20	1609	83	20

From Table I it is clear that the proposed HFSM with implicit modules is faster and requires smaller hardware resources than HFSMs with explicit modules.

The following details can be given for the resulting circuits that implement simple HGSs in Fig. 1 (block RAMs are not used for these circuits):

- HFSM with implicit modules with separation of modules and states (Fig. 3): $S=192$; $F_{max}=59.733$ MHz;
- HFSM with implicit modules without separation of modules and states (Fig. 2): $S=208$; $F_{max}=62.822$ MHz;
- HFSM with explicit modules: $S=266$; $F_{max}=51.135$ MHz;

The circuits, synthesized from HGSs in Fig. 1 and in Table 1, execute additional operations and the resources are needed not just for HFSM. Fig. 7 presents an example for which just HFSM implementation details are given below:

- HFSM with implicit modules with separation of modules and states (Fig. 3) with the total number of modules 4 and with the maximum number of states in modules 8: $S=74$; $F_{max}=73.888$ MHz;
- HFSM with implicit modules without separation of modules and states (Fig. 2) with the total number of states 24: $S=70$; $F_{max}=73.206$ MHz;
- HFSM with explicit modules: $S=89$; $F_{max}=50.339$ MHz.

For clarity of the results the encoding for stack memory, considered in [18], has not been done and such encoding enables the number of slices to be additionally reduced. The results clearly demonstrate that both models (see Fig. 2 and Fig. 3) are faster and they require smaller hardware resources than HFSM with explicit modules.

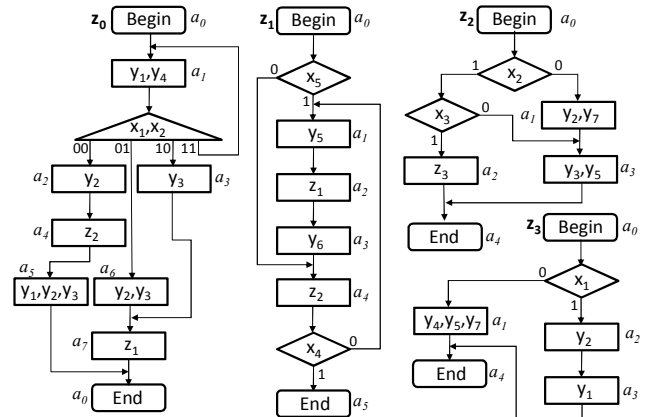


Figure 7. Example of HGSs z_0, z_1, z_2, z_3

From the experiments and the previous experience we found that hierarchy in hardware design is helpful when:

- A sequence of operations needs to be reused in different specifications. In such cases the sequence is assigned to a reusable module that is executed when necessary through hierarchical calls. For example, the mentioned above sorting algorithms are needed for priority buffers in [10] and, thus, these algorithms can be described by HGSs, implemented in HFSMs and reused for managing priorities.

- The strategy “Divide and conquer” is applied and makes it possible to cope with the growing complexity of digital circuits and systems. In this case HGSs are used to describe systems incrementally beginning with simple modules and continuing with creating more complicated modules from the modules that have already been developed, verified, and tested in hardware.
- A library of highly optimized and carefully tested reusable modules (HGSs) is available.
- Virtualization mechanisms are applied and overcome the problem of preliminary linkage by allowing macro operations (z_1, \dots, z_Q) to be defined during synthesis, and to be redefined later if necessary, after the relevant circuits have been designed.

The considered HFSMs are useful when we need:

- To convert easier software functions to hardware implementations through providing relatively similar mechanisms in software and in hardware;
- To accelerate execution of algorithms for implementing in hardware;
- To support easier the reuse technique; to simplify test and debug of hardware modules; to make refinement easier; to support modifiability, adaptation and virtual capabilities.

The proposed technique possesses the following distinctive features:

- The specification in form of HGSs is straightforward;
- HGSs are synthesizable and the relevant circuits can be easily implemented in hardware
- The technique makes it possible to support new features in hardware, such as recursion.

Since the HGSs and HFSMs enable design steps to be implemented through the selection and comparison of alternative circuits including virtualization, they are especially attractive for reconfigurable computing. This technique is also valuable for education because it permits more realistic student projects to be implemented and the results of such projects (HGS modules, in particular) can be easily reused by the next generations of students.

VI. CONCLUSION

The results of the paper clearly demonstrate that the proposed model of HFSM with implicit modules is faster and requires smaller hardware resources compared to HFSMs with explicit modules. The model incorporates a simple stack memory and in different modifications permits the known optimization methods developed for conventional FSMs to be entirely reused and the existing capabilities of HFSM to be inherited.

REFERENCES

- [1] G. de Micheli, *Synthesis and Optimization of Digital Circuits*, McGraw-Hill, Inc., 1994.
- [2] V. Sklyarov, *Synthesis of Finite State Machines Based on Matrix LSI*, Minsk, Science and Technique, 1984.
- [3] S. Baranov, *Logic Synthesis for Control Automata*, Kluwer Academic Publishers, 1994.
- [4] A. Girault, B. Lee, and E.A. Lee, “A Preliminary Study of Hierarchical Finite State Machines with Multiple Concurrency Models”, Memorandum No UCB/ERL M97/57, University of California, Berkeley, Aug., 1997.
- [5] V. Sklyarov, “Hierarchical Finite-State Machines and Their Use for Digital Control”, *IEEE Trans. on VLSI Systems*, vol. 7, no 2, pp. 222-228, 1999.
- [6] *Handbook of Nanoscience, Engineering, and Technology (Chapter on Hierarchical Finite State Machines and Their Use in Hardware and Software Design)*, edited by S.E. Lyshevski, G.J. Iafrate, W.A. Goddard, and D.W. Brenner, CRC Press, 2003.
- [7] C.A.M. Marcon, N.L.V. Calazans, and F.G. Moraes, “Requirements, Primitives and Models for Systems Specification”, *Proc. of the 15th Symp. on Integrated Circuits and Systems Design, Brazil*, 2002.
- [8] B. Aparicio del Moral, J.M. Jerónimo Zafra, J.F. Rodríguez Gómez, R. Sanz Mesa, R. Morales Muñoz, A. Rodríguez Trinidad, J.J. López Moreno, and the international Medusa team, “New Control System for Space Instruments. Application for Medusa Experiment”, *Proc. of the 7th Int. Planetary Probe Workshop, Barcelona, Spain*, June, 2010.
- [9] D.M. Muñoz, C.H. Llanos, M. Ayala-Rincón, and R.H. van Els, “Distributed approach to group control of elevator systems using fuzzy logic and FPGA implementation of dispatching algorithms”, *Elsevier: Engineering Applications of Artificial Intelligence*, 21, 2008, pp. 1309-1320.
- [10] V. Sklyarov, I. Skliarova, and A. Neves, “Modeling and Implementation of Automatic System for Garage Control”, *Proc. ICCAS-SICE’2009, Fukuoka, Japan*, August, 2009, pp. 4295-4300.
- [11] D. Harel, “Statecharts: A visual formalism for complex systems”, *Science of Computer Programming*, 8, June 1987, pp. 231-274.
- [12] S. Uchitel, J. Kramer, and J. Magee, “Synthesis of Behavioral Models from Scenarios”, *IEEE Transactions on Software Engineering*, vol. 29, Issue 2, February 2003, pp. 99-115.
- [13] J. Whittle and P.K. Jayaraman, “Generating Hierarchical State Machines from Use Case Charts”, *Proc. of the 14th IEEE International Requirements Engineering Conference, Minneapolis, USA*, Sept., 2006, pp. 16-25.
- [14] S. Lee, S. Yoo, and K. Shoi, “Reconfigurable SoC Design with Hierarchical FSM and Synchronous Dataflow Model”, *Proc. of the 10th Int. Symp. on Hardware/software codesign, Estes Park, USA*, May, 2002, pp. 199-204.
- [15] S. Ninos and A. Dollas, “Modeling recursion data structures for FPGA-based implementation”, *Proc. of the 18th International Conference on Field Programmable Logic and Applications – FPL’08, Heidelberg, Germany*, September 2008, pp. 11-16.
- [16] I. Skliarova and V. Sklyarov, “Recursion in Reconfigurable Computing: a Survey of Implementation Approaches”, *Proc. FPL’09, Prague, Czech Republic*, 2009, pp. 224-229.
- [17] V. Sklyarov and I. Skliarova, “Design and Implementation of Parallel Hierarchical Finite State Machines”, *Proc. HUT-ICCE’2008, Hoi An, Vietnam*, June, 2008, pp. 33-38.
- [18] V. Sklyarov, “Synthesis of Circuits and Systems from Hierarchical and Parallel Specifications”, *Proc. of BEC2010, Tallinn, Estonia*, Oct., 2010, pp. 37-48.
- [19] M. Koster and J. Teich, “(Self-)reconfigurable Finite State Machines: Theory and Implementation”, *Proceedings of DATE’2002, Paris*, 2002, pp. 559-566.
- [20] V. Sklyarov, I. Skliarova, and J. Lima, “Design and Implementation of Communicating Fixed and Variable Instruction Set Processors”, *Proc. of the 2nd International Conference on Computer and Electrical Engineering - ICCEE 2009, Dubai, UAE*, December 2009, pp. 163-167.
- [21] 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.
- [22] V. Sklyarov, “Reconfigurable models of finite state machines and their implementation in FPGAs”, *Journal of Systems Architecture*, 47, 2002, pp. 1043-1064.