Reconfiguration Technique for Adaptive Embedded Systems

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

Abstract-**This paper is dedicated to the design and implementation of adaptive embedded systems. Different synthesis methodologies are presented and discussed. Applying such methodologies the synthesis of circuits with support for modifiability and extensibility can be done. The paper describes: 1) specification of control algorithms for adaptive embedded systems and formal conversion of the specification to synthesizable VHDL code; 2) a model that is called a hierarchical finite state machine, which provides support for modularity, hierarchy, and recursion; 3) VHDL templates enabling the circuit to be synthesized in commercial CAD systems; and 4) the results of experiments.**

I. INTRODUCTION

Embedded systems are parts of larger systems and they are widely used in the manufacturing industry, in consumer products, in vehicles, in communication systems, in industrial automation, etc. Since typical embedded systems are heterogeneous and their specifications may change continuously we have to provide them with such characteristics as flexibility and extensibility. For many practical applications it is desirable to construct reusable components, such as algorithmic modules for logic control, etc. Adaptive embedded systems (AES) are capable to change their computational structure without adding physical components. In general, this can be achieved with the aid of reprogrammable devices such as field-programmable gate arrays (FPGA), which make possible to alter architectures of implemented circuits either statically or dynamically. However, in order to efficiently use capabilities of FPGAs it is necessary to suggests methods for the design of adaptive systems. AESs are very efficient for a large number of practical applications, such as network management, distributed systems, special-purpose co-processing, etc.

The proposed in this paper reconfiguration technique enables the designer to construct embedded systems with modifiable functionality through static and dynamic changes to the behavior of the advanced control unit. The latter is built in such a way that permits much more functions to be provided comparing to traditional finite state machines.

The following methods and models are used:

- Hierarchical specification of control algorithms in the form of hierarchical graph-schemes (HGS);
- Hierarchical finite state machines (HFSM);
- Support for recursive calls;
- Static and dynamic changes of HFSM functionality;
- Hardware description language templates.

The remainder of this paper is organized in 5 sections. Section II describes specification methods (HGSs) for adaptive control algorithms. Section III considers synthesis of HFSMs from HGS specifications. Section IV suggests adaptive models of HFSMs. The results of experiments are analyzed in section V. The conclusion is in section VI.

II. SPECIFICATION OF ADAPTIVE CONTROL ALGORITHMS

It is known [1] that hierarchical (in general) and recursive (in particular) algorithms can be described in the language called hierarchical graph-schemes (HGS). A HGS has the following formal description [1]:

- A HGS is a directed connected graph, which is composed of rectangular, rhomboidal and triangular nodes. Each HGS has one entry point which is a rectangular node marked with *Begin* and one exit point which is a rectangular node marked with *End*.
- The rectangular nodes contain either micro-instructions from the set $M_i=\{Y_1,Y_2,...\}$ or macro-instructions from the set $M_o=\{Z_1,Z_2,...\}$, or both. Any micro-instruction, Y_j, 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, such as loading a register or incrementing a counter. A macro-instruction incorporates a subset of macro-operations from the set $Z=\{z_1,...,z_Q\}$. Each macro-operation is described by a lower level HGS.
- Each rhomboidal node contains one element from the set X, where $X=\{x_1,...,x_L\}$ is the set of logic conditions. A logic condition is an input signal, which carries the result of a test, such as the state of a sensor.
- Each triangular node has one input and N outputs (N>2) permitting to provide N-directional transitions. It contains more than one element from the set X making it possible to activate the proper (only one) output.
- Inputs and outputs of the nodes are connected by directed lines (arcs) in the same manner as for an ordinary graph-scheme [2].

The considered specification method makes possible static and dynamic binding of modules (HGSs) to be provided. Dynamic binding is implemented through virtual macro-operations. This overcomes the problem of static linkage by allowing the control unit to define a macro-operation during synthesis, and then to redefine it later if necessary, after the control unit has been implemented. A macro-operation is called virtual, if it cannot be fixed during

the design. For any virtual element, which is described by the appropriate HGS, we can determine different implementations that depend on either an application-specific static installation of the respective control unit or dynamic re-configuration of the control in order to improve some application-dependent parameters. This gives us the following advantages:

- The ability to reuse previously constructed HGSs. By investing a little extra effort in the design, we can create a library of reusable components such as HGSs, which will facilitate the creation of similar products. Obviously any component can be designed and tested independently.
- The flexibility in the control algorithm in terms of possible trivial re-switching between relatively independent and simple components such as HGSs.
- The extension of a given control algorithm becomes a relatively simple matter. Indeed we can easily solve the problem of extending the behaviour of a HGS by modifying it.
- Adaptive facilities can be provided through selection of proper HGSs dependently on external events.

III. SYNTHESIS OF HFSMS FROM HGS SPECIFICATIONS

Advanced control algorithms with the considered above characteristics can be executed using the model of a HFSM. It is known [1] that the problem of hierarchical calls can be efficiently resolved using a stack memory. Two stacks in [3] (called *M_stack* and *FSM_stack*) permit to keep track for module invocations. The top register of the *M_stack* contains the code of the currently executing module. The top register of the *FSM_stack* is used as a memory for the currently executing module, i.e. it supplies states (codes of states) for any state transition required within the currently executing module. At the beginning, the top registers of the both stacks are set to the initial state a_0 of the initial module, which must be activated first according to the given algorithm. After that the following three allowed types of state transitions can be executed:

- A transition between states that belong to the same module. In this case the HFSM operates like an ordinary finite state machine (FSM).
- A transition to the first state of a next module z_n. In this case the operation *push("the code of the* z_n*")* is applied to the *M_stack* and the operation *push("the first state of the* z_n*")* is applied to the *FSM_stack*. This transition is known as *hierarchical call*.
- A transition from a currently executing module z_c to a module z_p from which the z_c was activated. In this case the operation *pop* is applied to the *M_stack* (thus, the top register of the *M_stack* will contain the code of z_p) and the operation *"pop + state transition"* is applied to the *FSM_stack* (thus, the *FSM_stack* will be switched to the code of the state *a(call* z_c*)*, from which the z_c was called, and a transition within the z_p will be executed). Note that it is necessary to avoid a repeated invocation of the module z_c from the state *a(call* z_c*)* during the operation *"pop + state transition"*. It is achieved in [3] through a special flag called "*return flag*". This third type of transition is known as *hierarchical return*.

The known approach [1,3] has 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.

Fig. 1 depicts the proposed structure of a HFSM which provides support for all features of the known model [1,3] but has just one simplified stack memory.

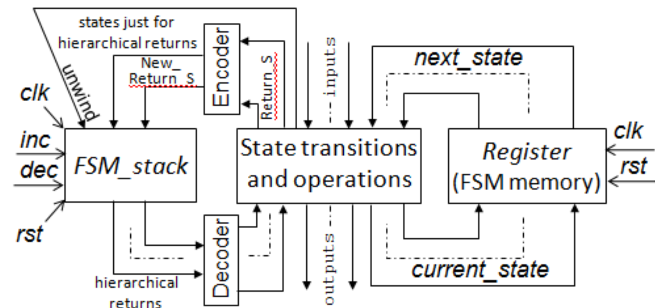

Figure 1. The proposed structure of a HFSM

There are three basic blocks in Fig. 1: a *register*; an *FSM_stack*; and a circuit that calculates next states for state transitions and executes the required operations. Thus, we will combine in the latter circuit a combinational part of FSM and the datapath for execution of operations. To avoid undesirable behavior let us synchronize the *FSM_stack* and the *register* using one clock edge (for example, *falling edge*) and the remainder block using another clock edge (for example, *rising edge*).

Synthesis of a HFSM with the structure in Fig. 1 includes the following steps:

1) Marking the given HGSs with labels that further will be considered as the HFSM states;
2) Customizing the proposed hardware description language (HDL) templates for all the blocks shown in Fig. 1 (VHDL will be used as HDL);
3) Synthesis of HFSM circuits from the customized VHDL templates using commercially available computer-aided design tools, such as ISE of Xilinx [4].

We propose different types of HGS marking that depend on the desired characteristics of HGSs and the circuits that are going to be synthesized. These types will be explained on examples shown in Fig. 2 and Fig. 3.

In the simplest case any called module (HGS) ($\rightarrow z$) cannot alter state transitions of the calling module ($z\rightarrow$) from the state $a(\rightarrow z)$ where $\rightarrow z$ is called. It means that all state transitions from the state $a(\rightarrow z)$ can be correctly determined before the module $\rightarrow z$ is called. Let us consider an example. The module (HGS) z_0 in Fig. 2 calls two other modules z_1 and z_2 in the states b_1 and b_2. We assume that the HGS z_0 has already been marked with the labels $a_0,\ldots,a_5,b_1,b_2$ and these labels can be seen as HFSM states (all necessary details will be given later). Hence, in Fig. 2 z_0 is a calling module $z\rightarrow$; z_1 and z_2 are called modules $\rightarrow z$.

Since state transitions from b_1 and b_2 in the HGS z_0 are unconditional, they cannot be influenced by the HGSs z_1 and z_2. Thus, the indicated above condition is satisfied.

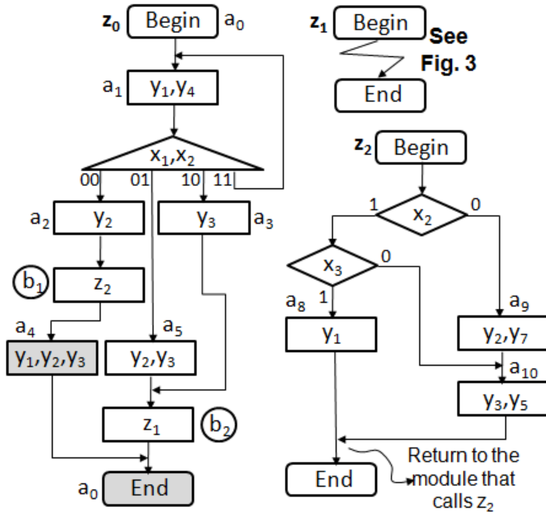

Figure 2. Examples of HGSs

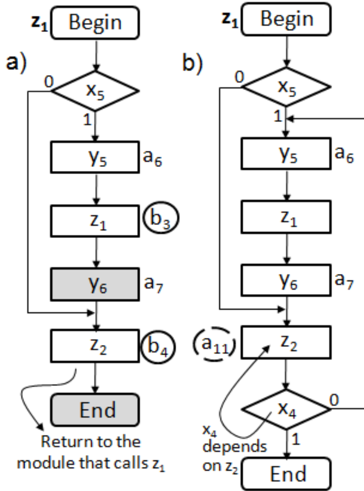

Figure 3. Different types of the HGS z_1

For HGS z_1 shown in Fig. 3,a the indicated above condition is also satisfied. Now let us consider HGS z_1 shown in Fig. 3,b. Suppose, the value of x_4 depends on the execution of the module z_2. In this case, the test of the input x_4 can be done just after the completion of the module z_2 and the value of x_4 is not known before the execution of the module z_2. Such more complicated case will be considered at the end of section III. If the value of x_4 does not depend on the execution of the module z_2, then our requirements are also satisfied.

For the HGSs shown in Fig. 2 and Fig. 3,a the marking is done as follows:

- The label a_0 is used for the node *Begin* and for the node *End* of the main HGS (the module z_0 in our example);
- All rectangular nodes (except remaining nodes *Begin* and *End*) have to be marked with different labels.

We will use for our example the labels $a_1,\ldots,a_{10},b_1,\ldots,b_4$. At the next step we have to describe all state transitions in VHDL templates. There are three templates for the three blocks, depicted in Fig. 1. The *register* is described similarly to a conventional FSM:

```
process(clk,rst)  -- clk is clock; rst is reset
begin
   if rst = '1' then  Rg <= a0;                 -- Rg is the register
   elsif falling_edge(clk) then Rg <= N_S; -- N_S is the next state
   end if;
end process;
```

The *FSM_stack* is described as follows:

```
process(clk,rst)
begin
if rst = '1' then  stack_ptr <= 0; FSM_stack(stack_ptr)<=(others=>'0');
elsif falling_edge(clk) then
   if inc = '1' then
         if stack_ptr = stack_size-1 then -- error handling
         else stack_ptr <= stack_ptr + 1;
               FSM_stack(stack_ptr) <= New_Return_S;
         end if;
    elsif dec = '1' then stack_ptr <= stack_ptr - unwind;
    else   null;
   end if;
end if;
end process;
```

Here *stack_ptr* is the stack pointer; the signal *inc* increments the *stack_ptr* and the signal *dec* decrements the *stack_ptr*. The signal *unwind* permits to execute *hierarchical return* through more than one level of HGSs. Let us consider an example. In Fig. 2 the module z_1 is called in the state b_2 and then in Fig. 3, a, z_1 calls z_2 in the state b_4. When the called module z_2 is terminated, the return has to be done to the module z_1 and then to the state from which the module z_1 was called. Thus, the *stack_ptr* has to be decreased by 2 (i.e. not by 1 as it is done normally). This is achieved using the line `stack_ptr <= stack_ptr-unwind;` in VHDL code of the *FSM_stack* and assigning *unwind* the value 2 in the state b_4. In other words, the value *unwind* has to be incremented in the states of modules $z_1,z_2,\ldots$ (except the main module z_0) that precede the node *End* and call new modules. The rules for such incrementing are described in [5]. The *FSM_stack* is needed to determine the next state after the called module ($\rightarrow z$) is terminated. For example, in Fig. 2 and Fig. 3,a z_2 is called in the state b_1 and b_4. After termination of z_2 (that was called in b_1) it is necessary to transit to the state a_4 and after termination of z_2 (that was called in b_4) it is necessary to transit to the state from which the module z_1 was called. Such state is known just in the calling module ($z\rightarrow$) and it is saved on the *FSM_stack* when a new module ($\rightarrow z$) is called. As soon as the module ($\rightarrow z$) is terminated, the proper state (let us call it return state or simply *Return_S*) is extracted from the *FSM_stack*. The number of return states is, as a rule, significantly less than the total number of states. In our example (Fig. 2 and Fig. 3, a) the return states are a_0, a_4 and the states for transitions after termination of the module z_1 (i.e. a_0 and a_7). Thus, totally there are just 3 return states (a_0, a_4, a_7) and they correspond to the rectangular nodes marked with gray color. Since the *FSM_stack* is needed just to keep return states, the

size of codes of such states is reduced by the *encoder* and then restored by the *decoder* in Fig. 1. It is done in the following fragment of VHDL, which permits to reduce the size of stack registers from 4 to 2 bits:

```
New_Return_S <= "01" when Return_S = a4 else
                "10" when Return_S = a0 else
                "11" when Return_S = a7 else "00";
```

Finally, the values of *New_Return_state* are saved in the FSM_stack instead of the values of *Return_S*.

VHDL template for the last block of Fig. 1 is the following:

```
process (rst, clk)
begin
if rst = '1' then -- initialization
elsif rising_edge(clk) then
  case RG is
    when a0 => -- transitions and operations for the state a0
    when a1 =>  -- transitions and operations for the state a1
         -- repeat for all other states a1,…,a10,b1,…,b4
    when others => null;
  end case;
end if;
end process;
```

Let us consider the full description of transitions and operations for some states:

```
when a1 =>  dec <= '0';  inc <= '0'; Y <= "0001001";
  case X(2 downto 1) is
        when "00" => N_S <= a2;
        when "01" => N_S <= a5;
        when "10" => N_S <= a3;
        when "11" => N_S <= a1;
        when others => null;
  end case;
when a2 => dec <= '0'; inc <= '0'; Y <= "0000010"; N_S <= b1;
when b1 => dec <= '0'; inc <= '1'; Y <= "0000000";
  Return_S <= a4; unwind <= 1;
  case X(3 downto 2) is
        when "01" => N_S <= a10;
        when "11" => N_S <= a8;
        when others => N_S <= a9;
  end case;
when a3 => dec <= '0'; inc <= '0';   Y <= "0000100"; N_S <= b2;
when b2 => dec <= '0'; inc <= '1';   Y <= "0000000";
  Return_S <= a0; unwind <= 1;
  if X(5) = '0' then N_S <= b4;  else N_S <= a6;  end if;
-- ………………………………………….
when b4 => dec <= '0'; Y <= "0000000";  inc <= '1'; unwind<=2;
  Return_S <= convert(conv_integer(FSM_stack(stack_ptr-1)));
  case X(3 downto 2) is
    when "01" => N_S <= a10;
    when "11" => N_S <= a8;
    when others => N_S <= a9;
  end case;
when a8 =>  dec <= '1'; inc <= '0';  Y <= "0000001";
   N_S <= convert(conv_integer(FSM_stack(stack_ptr-unwind)));
when a9 =>  dec <= '0'; inc <= '0';  Y <= "1000010"; N_S <= a10;
when a10 => dec <= '1'; inc <= '0'; Y <= "0010100";
   N_S <= convert(conv_integer(FSM_stack(stack_ptr-unwind)));
```

Note, that the labels $b_1,\ldots,b_4$ can be removed and when it is necessary to call new modules the proper transitions can be done from calling module to called module directly. It permits to reduce: 1) the number of states (from 15 to 11); 2) the number of clock cycles for executing the HGSs and respectively the execution time of the implemented algorithms. In this case the transitions from such states as a_2, a_3, a_5, a_6 and a_7 have to be done directly to the necessary states inside the modules z_1 and z_2 as follows:

```
when a2 => dec <= '0'; inc <= '1'; Y <= "0000010"; Return_S <= a4;
  case X(3 downto 2) is
    when "01" => N_S <= a10;
    when "11" => N_S <= a8;
    when others => N_S <= a9;
  end case;
when a3 => dec <= '0'; inc <= '1'; Y <= "0000100"; Return_S <= a0;
  if X(5) = '0' then unwind_i <= 2; unwind_d <= 2; Return_S <= a0;
    case X(3 downto 2) is
      when "01" => N_S <= a10;
      when "11" => N_S <= a8;
      when others => N_S <= a9;
    end case;
  else N_S <= a6; unwind_i <= 1; unwind_d <= 1;
  end if;
when a5 => -- the same lines as for the state a3
when a6 =>  dec <= '0'; inc <= '1'; Y <= "0010000"; Return_S <= a7;
   if X(5) = '0' then
     case X(3 downto 2) is
       when "01" => N_S <= a10;
       when "11" => N_S <= a8;
       when others => N_S <= a9;
     end case;
   else N_S <= a6;
   end if;
when a7 => dec <= '0'; Y <= "0100000";  inc <= '1';
   Return_S <= FSM_stack(stack_ptr-1); unwind_d <= 2;
   case X(3 downto 2) is
     when "01" => N_S <= a10;
     when "11" => N_S <= a8;
     when others => N_S <= a9;
   end case;
```

Now we can jump through more than one level to call a new module. For example, transitions from the state a_3 have to be done to the module z_1 and if x_5=0 to the module z_2 (to the states a_8, a_9 or a_{10}). That is why instead of one signal *unwind* two signals *unwind_i* (for *hierarchical calls*) and *unwind_d* (for *hierarchical returns*) have been used. Two lines of the *FSM_stack* are changed as follows:

```
stack_ptr <= stack_ptr + unwind_i;
stack_ptr <= stack_ptr - unwind_d;
```

Let us consider now the HGSs shown in Fig. 2 and Fig. 3,b, assuming that the value of x_4 depends on the execution of the z_2. In this case the rectangular node with z_2 in Fig. 3, b has to be marked with an additional label (*e.g.* a_{11}). Transition in the *register* (see Fig. 1) will never be done to the state a_{11} and, thus, we will not increase the number of cycles for state transitions. The state a_{11} will only be used as an indicator for *hierarchical returns*.

Let us consider a potential problem that can arise when a new module is called through more than one level. Suppose z_1 is called after the state a_3 and x_5=0. Thus, z_2 will be called instead of z_1 and the transition has to be done to either a_8, a_9 or a_{10}. Suppose after termination of z_2 the condition x_4=0. Thus, the return has to be done to the module z_1. In this case we have a transition through the module z_1 and the return to the module z_1. Therefore the proper return state in the module z_1 has to be explicitly indicated. It is achieved by replacing of the signal *Return_S* with an array of signals *Return_S()*. If a transition is done through a module then two elements of the *Return_S()* are assigned: one for the return state of the calling module (such as z_0) and one for the return state of the transit module (such as z_1). If there are many return states in the transit module than it is better to allocate the actual state (such as a_{11}) and to avoid transitions through intermediate modules. We believe that such cases require additional investigation and they are considered for future work.

For our example transitions from the states a_3 and a_5 will be executed as follows:

```
when a3 => dec <= '0'; inc <= '1'; Y <= "0000100"; Return_S(0) <= a0;
  if X(5) = '0' then unwind_i <= 2; Return_S(1) <= a6;
    case X(3 downto 2) is
      when "01" => N_S <= a10;
      when "11" => N_S <= a8;
      when others => N_S <= a9;
    end case;
  else N_S <= a8; unwind_i <= 1;
  end if;
```

The transitions from the states a_8 and a_{10} will be executed as follows:

```
when a10 =>  dec <= '1'; inc <= '0'; Y <= "0010100";
  case FSM_stack(stack_ptr-1) is
    when a11  =>
      if X(4) = '0' then N_S <= a6; unwind_d <= 1;
      else unwind_d <= 2; N_S <= FSM_stack(stack_ptr-2);
      end if;
    when others => N_S <= FSM_stack(stack_ptr-1);
  end case;
```

The code of *FSM_stack* has to be slightly modified in order to store the values of *Return_S(0)* for the calling module and the values *Return_S(1)* for the transit module.

Note that the technique considered above is also applicable to HFSMs with two stacks (i.e. *M_stack* and *FSM_stack*) considered in [1,3] and it also enables the number of clock cycles for execution of HGSs to be reduced.

IV. Adaptive Hierarchical Finite State Machines

In order to implement adaptive logic control algorithms using HFSMs, we have to provide the selection of modules dependently on some external events. This selection can be provided for both types of HFSMs considered. For the HFSM with *M_stack* and *FSM_stack* the selection can be done with the aid of block S shown in Fig. 4, which is based on RAM and can be programmed dynamically. As a result, we can call different versions of macro-operations from the same point of an HGS. Indeed, if the module $z_h \in Z$ has several versions, such as z_h^{v1}, z_h^{v2},... then we can replace one version z_h^{vi} with another version z_h^{vj} by changing the code of the module z_h^{vi} written at the address h of the block S, to the code of the z_h^{vj} written at the same address h. This gives us the following advantages:

- The ability to reuse previously constructed HGSs and previously designed HFSMs. By investing a little extra effort in the design, we can create a library of reusable components such as HGSs, which will facilitate the development of similar products. The basic reusable component is a separate HGS, which can be designed and tested independently.
- Flexibility in the control algorithm in terms of possible trivial re-switching between relatively independent and simple components such as HGSs.
- The extension of a given control algorithm becomes a relatively simple matter. Indeed, we can easily solve the problem of extending the behaviour of a HGS through modifying it.
- Different versions of macro-operations can be activated from the same point of a HGS.

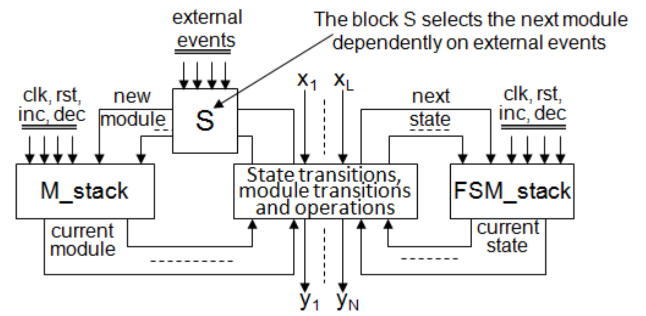

Figure 4. Adaptive HFSM based on *M_stack* and *FSM_stack*

For the HFSM with a single *FSM_stack* (see Fig. 1) the block S permits to properly select the first state of a module (HGS). For example, in order to change the module z_2 in the state b_1 (see Fig. 2) to a new module z_3, it is necessary to change transitions from b_1 leading to the module z_2 to transitions from b_1 leading to the module z_3 (see Fig. 5). For the example in Fig. 5 the VHDL code that describes transitions from b_1 has to be changed as follows:

```
when b1 => dec <= '0'; inc <= '1';
        Return_S <= a4; unwind <= 1;
        N_S <= a12;
```

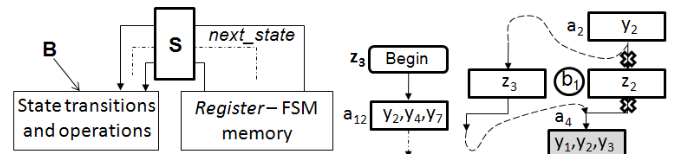

Figure 5. Adaptive HFSM based on a single *FSM_stack*

Note that to provide advanced adaptive control we have to be able not only to swap the modules but also to modify functionality of the modules. Functionality of any module is described in the block B (see Fig. 5). It is known that this functionality can be changed statically and dynamically

using RAM-based models [6]. This is achieved either by swapping pre-allocated areas on a chip in partially dynamically reconfigurable FPGAs, or by reloading memory-based cells in statically configured FPGAs using dual port capabilities. This can be done without introduction of additional clock cycles (a cascaded reprogrammable FSM proposed in [6] enables variable multidirectional state transitions to be realized during one clock cycle). Finally, it allows not only changing entry points to modules but also HGSs for modules. As a result, state codes of a new module can be the same as state codes of the removed modules. Note, that in many cases the model [6] requires a combinational part of HFSM and a datapath for execution of operations to be separated. Now let us summarize the proposed technique for adaptive HFSMs:

- The implemented algorithms have to be described as a set of modules (autonomous HGSs).
- The modules (HGSs) can be adapted to new conditions, which is achieved through a replacement of the existing modules (HGSs) with new (improved) modules (HGSs) applying the following methods: 1) modifying the implemented algorithms and the relevant RAM-based circuits [6] (reloading RAM blocks); 2) changing the entry points of the existing modules to entry points of the new modules in accordance with Fig. 4 and Fig. 5.

V. Experiments

The considered technique has been validated for a number of practical applications. The circuits have been synthesized in ISE 10.x of Xilinx [4] from specifications in VHDL (using the proposed templates) and implemented in commercially available FPGA xc3s500e-4fg320 of Spartan-3E family from Xilinx [4]. The experiments were done with the FPGA-based prototyping board NEXYS-2 of Digilent [7]. The following recursive algorithms have been described in HGSs, implemented in FPGA and tested: for discovering the greatest common divisor of integers; for data sorting; for solving combinatorial search problems (such as the SAT and matrix covering).

HFSMs considered above were also tested in the FPGA xc3s500e-4fg320 and the relevant circuits have the following implementation details:

i. HFSM for HGSs in Fig. 2 and Fig. 3, a with the states $a_1,\ldots,a_{10},b_1,\ldots,b_4$ (with *Encoder* and *Decoder*): the number of FPGA slices - 54; the maximum achievable clock frequency - 90 MHz;
ii. The same HFSM as for point *i* above, but without *Encoder* and *Decoder*: the number of FPGA slices - 65; the maximum achievable clock frequency - 94 MHz;
iii. HFSM for HGSs in Fig. 2 and Fig. 3, a with the states $a_1,\ldots,a_{10}$ (without *Encoder* and *Decoder*): the number of FPGA slices - 66; the maximum achievable clock frequency - 83 MHz;
iv. HFSM for HGSs in Fig. 2 and Fig. 3, b (without *Encoder* and *Decoder*): the number of FPGA slices - 67; the maximum achievable clock frequency - 71 MHz.

An analysis of the results of experiments permits the following conclusions to be drawn:

- HFSM with one stack (see Fig. 1) requires approximately 1.2 times less hardware resources than HFSMs with two stacks.
- The maximum attainable clock frequency of HFSMs with one and with two stacks is practically the same.
- Options *i* and *ii* above give the smallest hardware resources and the highest clock frequency, but it does not mean that the relevant HFSM is the fastest. This is because the HFSM for the options *i* and *ii* involves more clock cycles for execution of HGSs (such as that are shown in Fig. 2 and 3) than HFSMs for options *iii* and *iv*.
- Using the *encoder* and the *decoder* (see Fig.1) permits the FPGA resources to be reduced in about 1.2 times. However, the maximum attainable clock frequency is also slightly reduced.
- There is an opportunity for HFSMs with one stack to apply known optimization methods that have been developed for conventional state machines. HFSMs with two stacks are not so well suited for such optimization, mainly because states in different modules can be assigned the same codes.
- HFSMs with one stack also possess disadvantages, namely that modules become implicit and cannot be updated and refined easily. Although the HGSs for the new model are the same and all features are supported, modularity, hierarchy, and recursion become less clear at the implementation level.

VI. Conclusion

The technique, that allows modular modifiable circuits to be described, synthesized and implemented in hardware, has been proposed. It is shown that this technique permits to construct adaptive embedded systems, which are very useful for a number of practical applications. The following models, methods and tools have been discussed: a hierarchical specification in the form of hierarchical graph-schemes (HGSs); a hierarchical finite state machine that enables HGSs to be implemented in hardware; and synthesis of hardware circuits from HGSs. Applicability of the considered technique has been validated on examples implemented and tested in physical FPGA-based circuits.

References

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

[2] S. Baranov, *Logic Synthesis for Control Automata*. Kluwer Academic Publishers, 1994.

[3] 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.

[4] Available at http://www.xilinx.com.

[5] V. Sklyarov, I. Skliarova, "Recursive and Iterative Algorithms for N-ary Search Problems", International Federation for Information Processing, vol. 218, 2nd IFIP Symposium on Professional Practice in Artificial Intelligence - AISPP'2006, ed. J. Debenham, 19th IFIP World Computer Congress - WCC'2006, Santiago de Chile, Chile, August 2006, pp. 81-90.

[6] V. Sklyarov, “Reconfigurable models of finite state machines and their implementation in FPGA”, *Journal of Systems Architecture*, vol. 47, pp. 1043-1064, 2002.

[7] Available at: http://www.digilentinc.com.