Hierarchical Finite-State Machines and Their Use for Digital Control

Valery Sklyarov

Abstract—**This paper discusses the behavioral description, logic synthesis, and practical use of control units modeled as hierarchical finite-state machines with virtual states. The technique considered here provides a natural mechanism for top-down decomposition and enables us to develop any complex control algorithm step-by-step, where, at each stage, we are only dealing with a particular level of abstraction. Within any level, the specification encapsulates the control data and functions and allows recursive calls. Finally, the approach enables control units to be designed such that they incorporate new properties such as flexibility and extensibility. The primary functional components of a control algorithm can be reused in future applications.**

Index Terms— **Hierarchical finite-state machine, hierarchical specification, logic synthesis, modifiable control algorithm, recursive calls.**

I. Introduction

There are many kinds of devices that can be decomposed into a *datapath* and *control unit* [1], [2]. A datapath consists of storage units and combinational or functional units. A control unit performs a set of *instructions* by generating the appropriate sequence of *micro instructions* that depends on intermediate *logic conditions* or on intermediate states of the datapath.

In order to describe the behavior of a control unit, we can apply various forms of behavioral models [1]–[4]. We will use hierarchical graph schemes (HGS's) [5]–[7] for this purpose, which have the following formal description [5]. An HGS is a directed connected graph containing rectangular and rhomboidal nodes. Each HGS has one entry point, which is a rectangular node called *Begin*, and one exit point, which is a rectangular node called *End.* Other rectangular nodes contain either *micro instructions* or *macro instructions*, or both. Any *micro instruction* Y_j includes a subset of *micro operations* from the set $Y = \{y_1, \cdots, y_N\}$. A *micro operation* is an output signal, which causes a simple action in the datapath. Any *macro instruction* incorporates a subset of *macro operations* from the set $Z = \{z_1, \cdots, z_Q\}$. Each *macro operation* is described by another HGS of a lower level. We assume that each macro instruction includes just one macro operation. Each rhomboidal node contains one element from the set $X \cup \Theta$, where $X = \{x_1, \cdots, x_L\}$ is the set of *logic conditions*, and $\Theta = \{\theta_1, \cdots, \theta_1\}$ is the set of *logic functions*. A *logic condition* is an input signal, which communicates the result of a test. Each *logic function* is calculated by performing some predefined set of sequential steps that are described by an HGS of a lower level. Directed lines (arcs) connect the inputs and outputs of the nodes in the same manner as for an ordinary graph scheme [2].

Consider a set $E = Z \cup \Theta = \{\varepsilon_1, \cdots, \varepsilon_H\}$. Each element $\varepsilon_h \in E$ corresponds to an HGS Γ_h, which specifies either an algorithm for performing ε_h (if $\varepsilon_h \in Z$) or an algorithm for calculating ε_h (if $\varepsilon_h \in \Theta$). Let us assume that $Z(\Gamma_h)$ is the subset of macro operations and $\Theta(\Gamma_h)$ is the subset of logic functions that belong to the HGS Γ_h. If $Z(\Gamma_h) \cup \Theta(\Gamma_h) = \varnothing$, we have an ordinary graph scheme [2].

Using HGS's enables us to develop any complex control algorithm step-by-step, concentrating our efforts at each stage on a specified level of abstraction (i.e., on a particular element of the set E). Each component of the set E is usually very simple and can be checked and debugged independently. Fig. 1 shows an algorithm described by HGS's $\Gamma_1, \cdots, \Gamma_6$ (the symbols $a_0, \cdots, a_{26}$ will be used later).

The execution of an ordinary graph scheme based on a special traversal procedure was considered in [2]. We will use a similar approach that differs only in the interpretation of new complex operations such as $\varepsilon_h = z_q \in Z$ and $\varepsilon_h = \theta_i \in \Theta$. Each complex operation ε_h that is described by a separate HGS Γ_h has to be replaced with a new subsequence of operators that produces the result of executing Γ_h.

The main objective of this paper is to develop an approach to the design of virtual control devices that ensures that they have the properties of extensibility, flexibility, and reusability. It differs from other related work [8], [9], etc. in two basic directions. They are: 1) exploring a new method for the synthesis of control devices from a given set of HGSs and 2) developing a formal technique, which, on the one hand, ensures a correct implementation even in case of recursive calls and, on the other hand, allows benefits to be obtained from encapsulation in the hierarchical specification.

This paper is organized in seven sections. Section II discusses the hierarchical finite-state machine (HFSM), which has been proposed as a formal model of digital devices whose behavior is described by HGS's. Section III introduces an HFSM with virtual states. Section IV describes all the steps for HFSM synthesis. Sections V and VI explain the advantages of virtual HGS's, demonstrate their use for practical applications, and present the results of experimental tests. A conclusion is then presented in Section VII.

Manuscript received June 16, 1997; revised December 15, 1997 and March 30, 1998.

The author is with the Department of Electronics and Telecommunications/INESC-Aveiro, University of Aveiro, 3810 Aveiro Portugal.

Publisher Item Identifier S 1063-8210(99)02609-8.

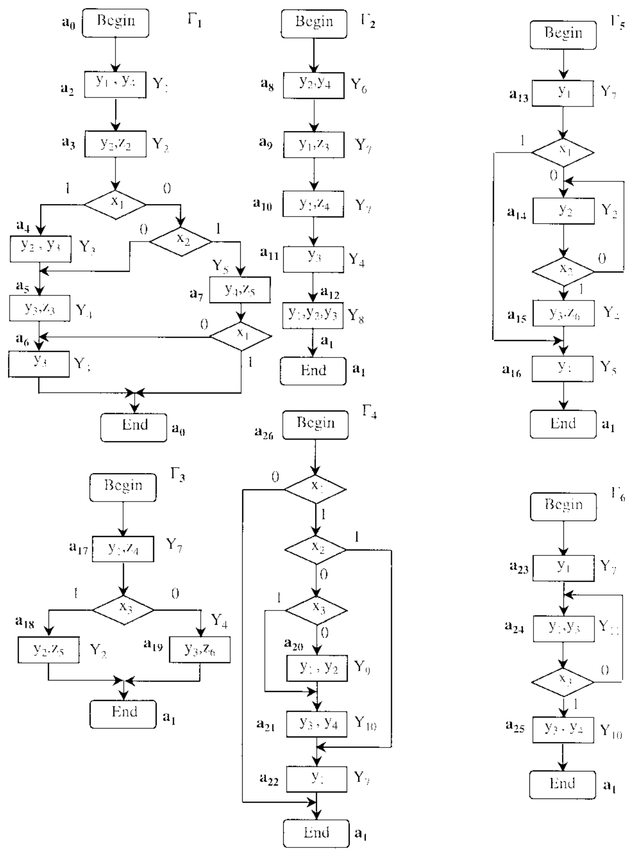


Fig. 1. Description of a control algorithm by HGS's.

II. HFSM

For a given set of HGS's, consider an undirected graph G_h [7], which shows a number of hierarchical levels and has a tree structure. The root $\boldsymbol{m}$ of the tree corresponds to the main HGS Γ_1 of level 1. The leaves of the tree correspond to HGS's, which do not contain elements from the set E (they are ordinary graph-schemes [2]). Consider the following sequence of HGSs: Γ_1 (level 1) $\Rightarrow \Gamma^2$ (HGS's of level 2) $\Rightarrow \Gamma^3$ (HGS's of level 3) $\Rightarrow \cdots$, where Γ^2 is a subset of the HGS's that are used to describe elements from the set $Z(\Gamma_1) \cup \Theta(\Gamma_1)$, Γ^3 is a subset of the HGS's that are used to describe elements from the sets $\cup_{\gamma \in \Gamma^2} Z(\gamma)$ and $\cup_{\gamma \in \Gamma^2} \Theta(\gamma)$. The same approach can be used to determine other subsets (Γ^4, Γ^5, etc.). Fig. 2(a) shows the graph G_h to be built for the set of HGS's given in Fig. 1.

For nonrecursive control algorithms, graph G_h can exhibit the maximum number of possible sequential calls of HGS's. However, for recursive algorithms, the depth of G_h is infinite. Consider another directed graph G_γ, which can be constructed for any control algorithm and enables us to eliminate possible future problem that might appear as a result of an infinite number of recursive calls. The G_γ has H vertices corresponding to the elements $\Gamma_1, \cdots, \Gamma_H$. Any vertex Γ_h is connected by directed lines to all vertices from the subset Γ^h. Fig. 2(b) depicts the graph G_γ to be built for the set of HGS's given in Fig. 1. If G_γ is an acyclic graph, the respective control algorithm is nonrecursive [6] and the maximum number of transitions σ is equal to the longest path on the graph G_γ. If

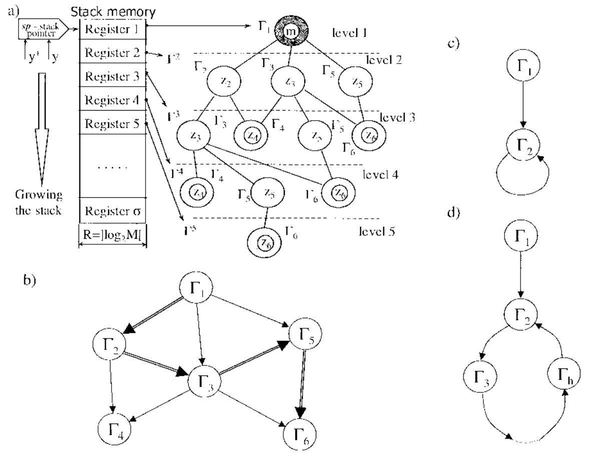


Fig. 2. Graph G_h. (a) Utilizing the stack memory. (b) Deciding the required size of the stack memory. (c), (d) Examples of recursive calls.

G_γ is a cyclic graph [see examples in Fig. 2(c) and (d)] the control algorithm is recursive and the problem becomes more complex. In this case, an HGS may call itself directly [see Fig. 2(c)] or indirectly [see Fig. 2(d)]. Recursion brings the same problem here as it does in software engineering, i.e., the process of repeated calls might be infinite. However, recursive algorithms are very often more compact and might be easier to design and understand than the nonrecursive equivalents. For a correct recursive algorithm, the problem is resolving step-by-step in such a way that we will finally reach a step where the recursive cycle will be broken, i.e., the next sub-steps will not be recursive. In this case, the number of hierarchical levels σ is finite. The algorithm can be implemented in hardware with a given stack memory if $\sigma \leq \Sigma$, where Σ is the number of registers in the stack memory. The problem is how to find out the value of σ. If G_γ is an acyclic graph, finding the longest path of G_γ [1] can give a solution. For instance, double arrow lines in Fig. 2(b) show the longest path. If G_γ is a cyclic graph, we have to examine the control algorithm together with the relevant datapath. In the general case, this problem cannot be solved theoretically because the datapath might receive data from other resources in the system, which might be unknown. However, for many practical applications, it is feasible to specify some constraints, such as the maximum number of data records that can be handled, etc.

In Fig. 2(a), $\Gamma^2 = \{\Gamma_2, \Gamma_3, \Gamma_5\}$, $\Gamma^3 = \{\Gamma_3, \Gamma_4, \Gamma_5, \Gamma_6\}$, $\Gamma^4 = \{\Gamma_4, \Gamma_5, \Gamma_6\}$, $\Gamma^5 = \{\Gamma_6\}$, and $m = z_1$ is the main part of the algorithm. Micro operations $\boldsymbol{y}^+$ and $\boldsymbol{y}^-$ are used to increment and to decrement, respectively, the stack pointer (sp). The problem of switching to various levels can be efficiently resolved using an HFSM with a *stack memory* [5], [6] (see Fig. 3). At the beginning, the top of the stack is the register that is used as the HFSM memory for the HGS Γ_1. Suppose it is necessary to perform an algorithm for a component $\varepsilon_h \in Z(\Gamma_1) \cup \Theta(\Gamma_1)$. In this case, we increment the sp by activating a special micro operation $\boldsymbol{y}^+$ and set the new register that is now located on the top of the stack into the first state for Γ_h. As a result, the previous top register of the stack stores the interrupted state of Γ_1 and the new top register of the stack stores the state of the entry point for Γ_h.

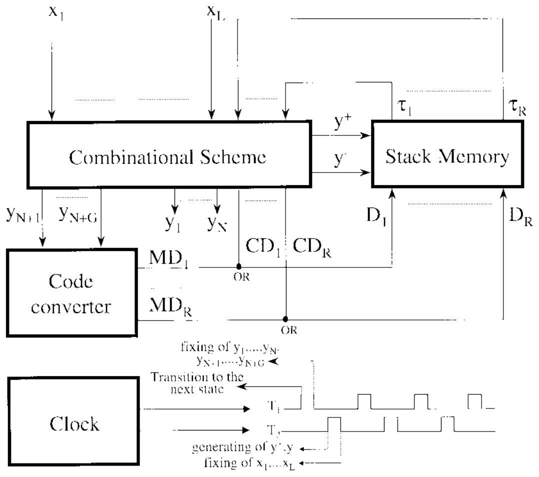


Fig. 3. Basic structure of the HFSM.

The same sequence of steps can be applied to other levels. The total size of the stack Σ is determined in such a way that the correctness of hierarchical control algorithm is assured. When the execution of an HGS of level h is terminated, we will perform the reverse sequence of steps to return back to the interrupted HGS. In this case, we decrement the sp by activating a special micro operation $\boldsymbol{y}^-$.

The stack memory is used to keep track of the calls of any macro operations (logic functions). The code converter (CC) has an input register which stores values of $y_{N+1}, \cdots, y_{N+G}$, where G is the minimum number of bits required to represent the binary codes for the HGS's for a given control algorithm (see Section IV for details). The values of $y_{N+1}, \cdots, y_{N+G}$ are the bits for the code to select the HGS to be executed next. This code is converted to produce the first state of Γ_h, which appears on the outputs $MD_1, \cdots, MD_R$. Here $D_1 = CD_1 + MD_1, \cdots, D_R = CD_R + MD_R$ (+ is the logic OR operation).

If there are no transitions from one HGS to another, the HFSM operates like an ordinary finite-state machine (FSM). If it is necessary to call a new HGS Γ_h in order to perform either a *macro operation* or *a logic function*, the following sequence of actions will be carried out. The code $K(\varepsilon_h) = (e_{h1} \cdots e_{hG})$ of the Γ_h is generated and stored in the input register of the block CC. Let us agree for simplicity that $K(\varepsilon_h)$ is the binary code of h and $e_{hg} \in \{0, 1, -\}, g = 1, \cdots, G$, $-$ is a don't care value. The sp is incremented $(y^+ = 1)$ and, as a result, a new register RG_{new} of the stack memory will be selected as the current register of the HFSM. The previous register $RG_{\text{new}-1}$ stores the state of the HFSM when it was interrupted. The code $K(\varepsilon_h)$ is converted to the code of the first state of the HGS Γ_h, which is generated on the outputs $MD_1, \cdots, MD_R$ of the block CC (at this time $CD_1 = \cdots = C_R = 0$). Now the HGS Γ_h is responsible for control from this point until it is terminated. After termination of Γ_h, the micro operation $\boldsymbol{y}^- = 1$ is generated in order to return to the interrupted state. As a result, control is passed to the state in which we called the HGS Γ_h. The subsequent actions are performed in the same manner as for an ordinary FSM.

Outputs $(CD_1, \cdots, CD_R)$ and $(MD_1, \cdots, MD_R)$ affect required transitions in different periods of time. That is why we need either OR (wired OR) or XOR operation for the transition functions $D_1, \cdots, D_R$.

It should be mentioned that we do not change the definition of the FSM [1], [2]. It may be defined as a sextuple vector $S = (A, X, Y, \psi, \varphi, a_0)$, where $A = \{a_0, a_1, \cdots, a_M\}$—is a finite set of states, $a_0 \in A$ is an initial state, $X = \{X_1, X_2, \cdots\}$ is a finite set of input vectors, $X_i = (x_1, \cdots, x_L)_i, x_l \in \{0, 1, -\}, Y = \{Y_1, Y_2, \cdots\}$—is a finite set of output vectors, $Y_j = (y_1, \cdots, y_N)_j, y_n \in \{0, 1, -\}, \psi$: $A \times X \rightarrow A$ is a transition function, mapping a subset $A \times X$ onto a subset of A. This function defines a next state $a(t+1) \in A$ depending on the current state $a(t) \in A$, and an input vector $X(t) \in X$: $a(t+1) = \psi(a(t), X(t))$. φ is an output function, which allows us to define output vectors $Y(t)$ from the set Y. HFSM can be described as follows:

$$
\begin{aligned}
a(t+1) &= \psi(a(t), X(t), z(t), \theta(t)) \\
Y(t) &= \varphi(a(t), X(t)) \\
Y(t) &= \varphi(a(t)).
\end{aligned}
$$

However, because $z(t)$ and $\theta(t)$ are calculated for the current state $a(t)$, we can formally convert these equations to the equations considered above. As a result, we can describe a control algorithm by a set of HGS's and then convert the given description to a state transition table which we can use to synthesize the combinational scheme (see Fig. 3) by invoking known methods of logic synthesis. After that, we can connect the combinational scheme to the predefined blocks such as the CC and the stack memory. Thus, we will design an HFSM which behaves in accordance with the given description.

The stack memory is used to manage the hierarchy and does not affect transitions. When we want to generate a macro operation z_q in a state a_m, we activate the increment micro operation y^+. As a result, we select the next level of the stack and set the new register to the first state a_s of z_q. In this case, $a(t) = a_m a(t+1) = a_s, z(t) = z_q = F_z(a(t), X(t)), F_z$ is a conversion function $a(t+1) = \psi(z(t)) = \psi(F_z(a(t), X(t)))$. Finally, the next state $a(t+1)$ depends on the current state $a(t)$, and a current input vector $X(t)$. When the macro operation z_q is terminated, we generate the decrement micro operation y^-. As a result we select the previous level of the stack which contains the state a_m. In this case, we perform an ordinary transition $a(t) = a_m, a(t+1) = \psi(a(t), X(t))$.

It should be mentioned that a stack memory is also widely used in the hardware implementation of subroutine calls in ordinary computers. However, in this case, it mainly keeps track of the calls and it does not manipulate the currently active states of any executable procedure. A program counter handles these states. In case of an HFSM, the stack registers play an active role in the execution of the control algorithm, i.e., they take part in state transitions. The technique considered allows the stack memory to be affected in the same manner as for an ordinary FSM, i.e., the hierarchical specification does not require any explicit definition of the operations for a hierarchical implementation. The process of stack memory management is common for any HFSM and the same stack is shared among multiple communicating HGS's. This process is hidden and is supported by local logic incorporated into the primary components (see Fig. 3) and reused for any control circuit.

It should be noted that compared to an ordinary FSM, an HFSM has a more complicated synchronization. Actually, there are many acceptable kinds of synchronization, and we want to suggest just one of them [7]. The selection of the proper HGS can only be done in states containing macro operations and logic functions. In states where a new HGS is selected, all outputs $CD_1, \cdots CD_R$ (see Fig. 3) are set to zero. In states where a previously called HGS is terminated, all outputs $MD_1, \cdots MD_R$ are set to zero. The sp can only be decremented in a single specially allocated state a_1. The outputs y^+ and y^- can only be generated in different states. At the beginning, we provide initial settings of all registers to zero. Fig. 3 illustrates possible waveforms for discrete synch signals according to the mode we have just considered.

III. An HFSM with Virtual States

In Section II, we assumed a preliminary binding between HGS's in a given set. This means that all links between different HGS's have been resolved during the process of the control-unit synthesis. The block CC has been programmed to convert the codes $K(\varepsilon_1), \cdots, K(\varepsilon_H)$ to the initial states of the HGS's $\Gamma_1, \cdots, \Gamma_H$. In order to provide extensibility, flexibility, and reuse of the scheme in Fig. 3, we have to exclude hardwired links between relatively independent components of the control algorithm, such as the HGS's.

In order to provide links that we can alter, let us introduce virtual macro operations (logic functions) that overcome the problem of preliminary linkage by allowing the control unit to define a macro operation during synthesis, and redefine it later if necessary, after the control unit has been designed. Borrowing some of the ideas of object-oriented programming [4], we can even consider the possibility of pure virtual macro operations, which have not been implemented during synthesis at all (however, we can implement them later). They are described by an HGS containing just two nodes following each other: *Begin* and *End.*

If we want to provide dynamic binding, the block CC will have the structure of random access memory (RAM). A virtual macro operation z_q^v, to be indicated by the superscript v, is described by a virtual HGS. It denotes that we are not able to predict and, as a result, to hardwire a direct link between z_q^v and the proper HGS. Depending on a particular situation, for a given z_q^v, we can execute one of many different HGS $\Gamma_q^1, \Gamma_q^2, \cdots$. A concrete binding is resolved after the HFSM has been designed. In order to provide hardware support for it, we have introduced an HFSM with *virtual states*. A state is called virtual if the hierarchical transition(s) from it have not been fixed in the given HGS.

IV. Synthesis of an HFSM

The problem of synthesis is as follows. For given control algorithm Λ, described by the set of HGS's, construct the FSM that implements Λ. Consider how to solve this problem for a Moore machine.

The synthesis includes the following steps:

1) transforming the HGS's to a *state transition table*;
2) state encoding;
3) combinational logic optimization and design of the final scheme.

The first step is divided into three sub steps, which are: 1) marking the HGS's with labels; 2) recording all transitions between the labels in the *extended state transition table*; and 3) converting the extended table to ordinary form [2].

In order to mark given HGS's, it is necessary to do the following (see Fig. 1). The label a_0 is assigned to the node *Begin* and to the node *End* of the HGS Γ_1. The label a_1 is assigned to all nodes *End* of the HGS's $\Gamma_2, \cdots, \Gamma_H$. The labels $a_2, a_3, \cdots, a_M$ are assigned to: 1) unmarked rectangular nodes in all HGSs; 2) inputs of rhomboidal nodes with logic functions, which directly follow either other rhomboidal nodes or rectangular nodes with macro operations; 3) the *Begin* nodes of the HGS's $\Gamma_2, \cdots, \Gamma_H$ connected to a rhomboidal node with logic condition; and 4) the input of a rhomboidal node with a logic condition, which follows a connection from the *Begin* node of the HGS Γ_1. Repeating labels (apart from a_1) in different HGS's is not allowed.

Now, the labels $a_0, \cdots, a_{M-1}$ are considered to be HFSM states. In order to build the extended state transition table, it is necessary to perform the following actions. Record all transitions $a_m X(a_m, a_s) a_s$, where $a_m \in \{a_2, \cdots, a_M\}$ $(a_m \neq a_0, a_m \neq a_1), a_s \in \{a_0, \cdots, a_M\}, X(a_m, a_s)$ is a product of input variables (logic conditions) and logic functions, which causes the transition from a_m to a_s. For any unconditional transition, $X(a_m, a_s) \equiv 1$. Record all transitions $a_0 a_s, a(\varepsilon_h) a_k$ where a_s is the label for a node (for an input) to which there is a direct connection from the *Begin* node of the HGS Γ_1, $a(\varepsilon_h)$ is a node (input of a node) containing the element ε_h, a_k is the label for a node, which starts the HGS for the element ε_h. Write down the micro operations $Y(a_m)$ and macro operations $Z(a_m)$ generated in the rectangular node marked with the label a_m. Record the transitions considered in the extended table.

In order to convert the table from an extended to an ordinary form, it is necessary to do the following. Encode the HGS's $\Gamma_1, \cdots, \Gamma_H$. Suppose for simplicity that the code $K(\Gamma_h)$ of the HGS Γ_h is the binary value of h with minimal length G. If a set $Z(a_m)$ contains a macro operation $\varepsilon_h = z_q \in Z$, then it has to be replaced with new output variables that are y^+ and $y_{N+1}, \cdots, y_{N+G}$. We are only choosing variables y_{N+g} for which $e_{hg} = 1$ (the e_{hg} is a value of the bit number g in the code of Γ_h). Suppose a_u is an input label of a rhomboidal node. If the node contains a logic function $\theta_i = \varepsilon_h$, we must add y^+ and the respective variables $y_{N+1}, \cdots, y_{N+G}$ to the set $Y(a_u)$. If an input of the rhomboidal node containing a logic function θ_i has not been marked, we must add y^+ and the respective variables $y_{N+1}, \cdots, y_{N+G}$ to the sets $Y(a_{f1}), Y(a_{f2}), \cdots$, where the states $a_{f1}, a_{f2}, \cdots$ have been inserted to mark the nodes that are the predecessors of the considered rhomboidal node with the logic function θ_i. The calculated value of a logic function is stored in a specially allocated storage element and then used as a normal logic condition. The final table describes the hierarchical transitions implicitly and can be visualized as an ordinary table.

In order to program the CC, we have to perform the following steps. Record the transition from the state a_0 assigned to the *Begin* node of the HGS Γ_1. For our example in Fig. 1, this is $a_0 \Rightarrow a_2(\Gamma_1)$. We put the name of HGS that is the owner of the state in parentheses. Record all transitions between different HGS's. For our example in Fig. 1, they are $a_3 \Rightarrow a_8(\Gamma_2)$, $a_5 \Rightarrow a_{17}(\Gamma_3)$, $a_7 \Rightarrow a_{13}(\Gamma_5)$, $a_9 \Rightarrow a_{17}(\Gamma_3)$, $a_{10} \Rightarrow a_{26}(\Gamma_4)$, $a_{15} \Rightarrow a_{23}(\Gamma_6)$, $a_{17} \Rightarrow a_{26}(\Gamma_4)$, $a_{18} \Rightarrow a_{13}(\Gamma_5)$, $a_{19} \Rightarrow a_{23}(\Gamma_6)$. For all transitions $a_m \Rightarrow a_s(\Gamma_j)$ considered above, convert $K(\Gamma_j)$ to a_s. If we assume for simplicity that the code of a_s is the binary value of the index s with minimal length $R = \text{int} \log_2 M$, then our conversions are as follows: $K(\Gamma_1) = 001 \Rightarrow a_2 = 00010$, $K(\Gamma_2) = 010 \Rightarrow a_8 = 01000$, $K(\Gamma_3) = 011 \Rightarrow a_{17} = 10001$, $K(\Gamma_4) = 100 \Rightarrow a_{26} = 11010$, $K(\Gamma_5) = 101 \Rightarrow a_{13} = 01101$, $K(\Gamma_6) = 110 \Rightarrow a_{23} = 10\,111$. For all $a_m \Rightarrow a_s(\Gamma_j)$, the block CC converts $K(\Gamma_j)$ to a_s. Now we can synthesize the complete scheme of the HFSM using known methods ([1], [2], etc.).

V. Virtual HGS's and Their Use for Practical Applications

Suppose that all elements of the set $E = \{\varepsilon_1, \cdots, \varepsilon_H\}$ are virtual, which means that all the given HGS's $\Gamma_1, \cdots, \Gamma_H$ are also virtual. As a result, we can call different versions of macro operations (logic functions) from the same point of an HGS.

There are two acceptable and reasonable ways in which the binding of any modifiable connection might be established. These are before run time and during run time. The first kind of binding is called static because it is established outside the scope of dynamic physical control. Applying static binding limits the future potential for modifications of predefined control algorithms. Dynamic binding establishes links during run time. This means that while one HGS is executing, another HGS might be either being modified or replaced. Any two HGS's can be swapped by setting a new entry point in the RAM-based CC.

Let us have a look at an example that demonstrates the advantages of the proposed architecture, such as capability for recursive calls, as well as the extensibility and flexibility.

Consider a binary tree whose nodes contain three fields that are a pointer to the right child node, a pointer to the left child node, and a value (let us say an integer or a pointer to a string). The nodes are maintained so that, at any node, the left sub-tree contains only values that are less than the value at the node, and the right sub-tree contains only values that are greater (see [10] for details). All nodes are accommodated in RAM (see Fig. 4) and we want to design hardware that can be customized to solve one of the following problems:

1) determining the maximum value;
2) determining the minimum value;
3) sorting the values in descending sequence;
4) sorting the values in ascending sequence.

One potential implementation is shown in Fig. 4.

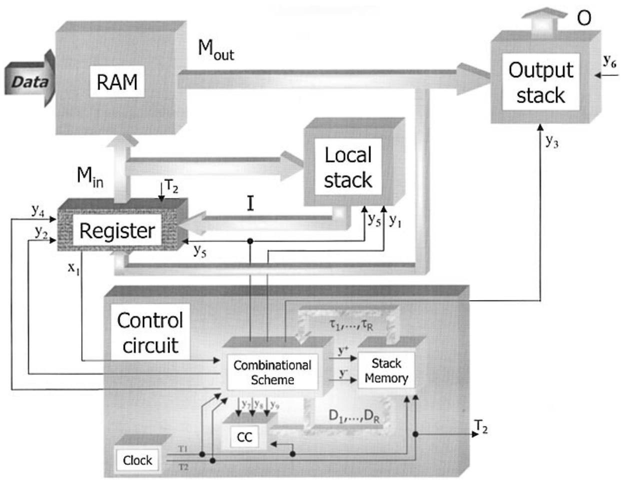


Fig. 4. Hardware implementation for operations using a binary tree.

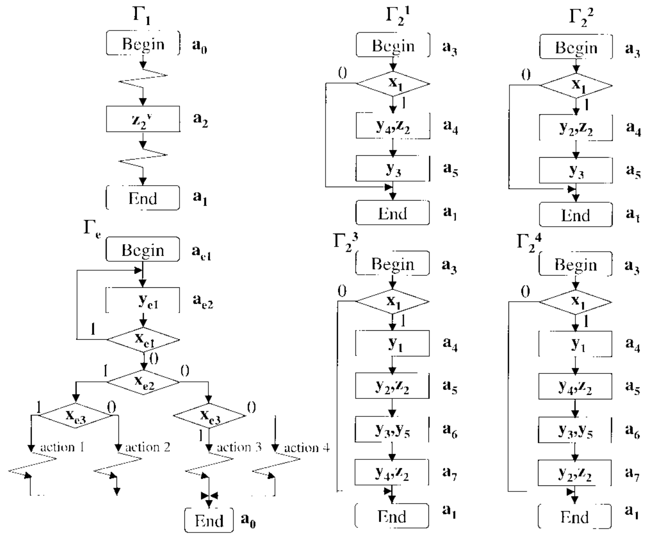


Fig. 5. HGS's specifying the control unit in Fig. 4.

Consider the set of HGS's depicted in Fig. 5. The first HGS Γ_1 can perform various operations (they are indicated by zigzags in Fig. 5). When required, it calls an HGS Γ_2 via the virtual macro operation z_2^v. The latter might have several different implementations, such as: 1) Γ_2^1 for finding a maximum value; 2) Γ_2^2 for finding a minimum value; and 3) Γ_2^3 and Γ_2^4 for finding the ordered sequence of values from maximum to minimum or vice versa. All HGS's $\Gamma_2^1 - \Gamma_2^4$ are recursive. A single logic condition x_1 allows to find the nodes that do not point to other nodes. Micro operations $y_1, \cdots, y_6$ force the following actions in the datapath:

y_1	push data onto the local stack;
y_2	calculate an address of the left node from the currently active node;
y_3	push data onto the output stack;
y_4	form the address of the right node from the currently active node;
y_5	pop data from the local stack and store it in the register;
y_6	pop data from the output stack to external output O (see Fig. 4).

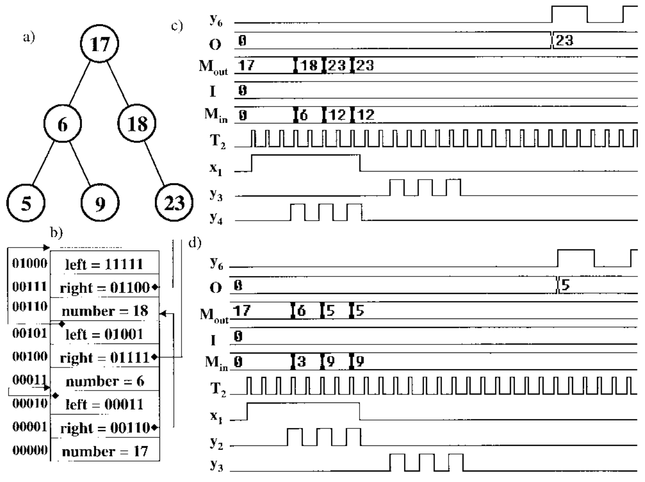


Fig. 6. (a) Binary tree for our example. (b) Example of contents of RAM in Fig. 4. (c) Waveforms illustrating how to find the maximum value. (d) Waveform illustrating how to find the minimum value.

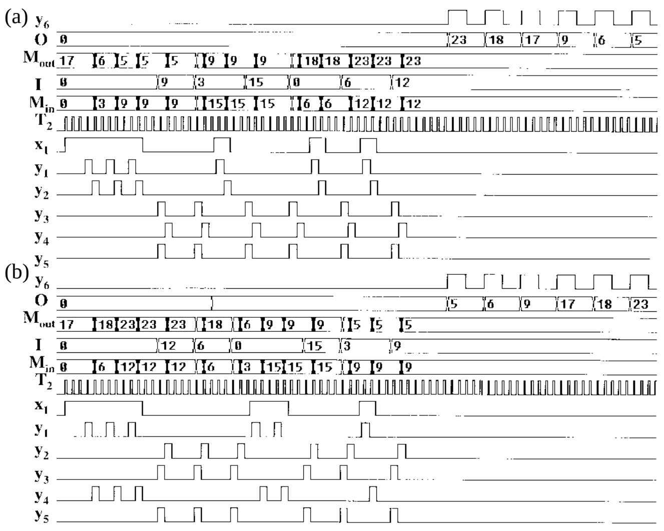


Fig. 7. (a) Waveforms illustrating the results of sorting values in ascending sequence and (b) in descending sequence.

In order to define the size of the stack memory that is used for the control circuit, we have to set some predefined constraints for the datapath in Fig. 4, such as the maximum permitted path from the root of the binary tree. If this is either problematical or unclear, it is reasonable to specify an exception, such as stack overflow, and an HGS that will handle this exception (see the HGS Γ_e in Fig. 5).

The entry point of Γ_e is permanently attached to the level Σ of the stack. Any attempt to reach this level causes an exception to be thrown, i.e., the HGS Γ_e will be automatically executed. It might set a special indicator (see micro operation y_{e1} in Fig. 5) asking what to do. The higher level control system tests this indicator and responds through logic conditions specified within the HGS Γ_e (see rhomboidal nodes x_{e1}, x_{e2}, x_{e3} in Fig. 5). This information is used to perform actions that allow the system to cope with the problem, which, for instance, could be any of the following:

1) set the stack to the initial state;
2) extend the stack and continue the task;
3) switch to a new stack memory and restart the task, indicate an unrecoverable error, etc.

The scheme in Fig. 4 was described in the VHDL language. The results of the simulation for the binary tree, depicted in Fig. 6(a), are shown in Figs. 6 and 7. Suppose that for given set $\Gamma_2 = \{\Gamma_2^1, \Gamma_2^2, \Gamma_2^3, \Gamma_2^4\}$ the desired HGS has to be selected during run time, depending on what operation we want. In order to do this, we can dynamically modify the given control algorithm, i.e., the macro operation z_2^v will be associated with the appropriate HGS during the execution of the HGS Γ_1. In addition, we can extend the set Γ_2 on the fly if necessary by reconfiguring our scheme with the aid of a high-level control system.

In order to accomplish the required modification, we can use two different techniques. In the first case, during synthesis we provide direct mapping of HGS's to modifiable elements (ME's) (see the left-hand side and bottom of Fig. 8) that are either reconfigurable, such as components implemented within an FPGA, or reprogrammable, such as elements with a regular matrix structure. If we want to modify/replace any HGS (e.g., the right-hand-side bottom HGS in Fig. 8), we have to modify/replace the associated ME (i.e., we might replace ME_V with ME_h in Fig. 8).

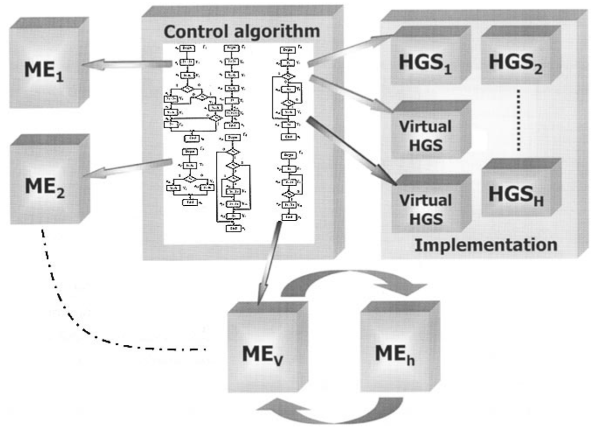


Fig. 8. Dynamic modifications of the control algorithm.

In the second case, the direct mapping considered above has not been provided. However, we can implement all the required additional HGS's in hardware (see the right-hand side of Fig. 8). If we want to replace an HGS, we load a new entry state for the new HGS in the CC (see the right-hand side of Fig. 8).

Finally, the approach considered gives the following advantages. It enables flexibility and extensibility in the control algorithm and makes it possible for previously constructed HGS's and previously designed HFSM's to be reused [11]. It provides an exception handling mechanism. At least two kinds of exceptions have to be taken into account: errors and reload conditions. The latter enables us to support execution of virtual algorithms, i.e., the capability to reload HGS's during run time.

TABLE I
EXPERIMENTAL RESULTS

name of examples	number of nodes	L/N (in/out)	H	depth of stack	size
Sprin	34	3/4	5	4	44
Kluw	39	8/14	3	4	59
PapGS	46	3/4	5	5	62
Sort	49	6/20	1	exc	61
Bar	51	11/21	3	2	68
Gren	56	6/12	3	3	97
H_sched	75	15/33	5	2	111
ALU_a2	105	5/42	3	2	152
Synch	120	14/9	15	4	165
ALU_a1	189	14/49	20	3	261
AbGS	504	96/116	27	4	776

VI. Experimental Tests

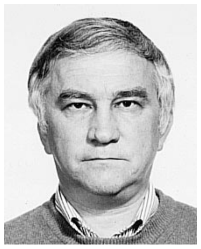
The proposed architecture of an HFSM (see Fig. 3) was described in VHDL and has been carefully examined for different control algorithms (we used the V-system VHDL analyzer). The results of the test have proven that the target requirements of the HFSM, such as extensibility, flexibility and reuse, have been satisfied. The best results were obtained for recursive algorithms. Since the proposed technique enables us to convert a given set of HGS's to an ordinary state transition table, we can further apply a variety of known methods of logic synthesis. Consequently, after the first step (see Section IV), we can use any synthesis tool that is available for an ordinary FSM with the same time constraint and memory requirement. The first synthesis step was carefully tested (see Table I, where L is the number of inputs, N is the number of outputs, and H is the number of macrooperations and logic functions). For all examples from Table I, the time for step 1 did not exceed 1 s on a 200-MHz PC Pentium. In the last column, we have shown the size of alterable part of combinational scheme (see Fig. 3) of an HFSM in gate equivalents, such as components of Xilinx PRIMS library for FPGA XC6200 (steps 2 and 3 were carried out using methods similar to [2]). For one example, the size of stack can be determined only at execution time and an HFSM has an exception handler (it is denoted in Table I by exc). Gaining the advantages considered above gave rise to a number of supplementary tasks, which required additional time and effort. For example, it was necessary to determine the size of stack, to design the additional HGS's that provide exception handling, etc. It should also be noted that the size of the stack memory is larger than the size of the register for an ordinary FSM, especially when we use recursive calls. However, the size of the combinational part is usually smaller and the best results were obtained when recursive calls were used. Actually, the latter can also be emphasized as a significant advantage of the proposed architecture.

VII. Conclusion

We have attempted to develop an approach to the synthesis of an FSM based on a hierarchical behavioral specification. The proposed technique can be used to design a wide range of control devices and it provides them with such characteristics as *extensibility* and *flexibility*. The developed model directly supports top-down decomposition and allows recursive and incompletely specified calls. The entire control algorithm is described by a set of HGS's that can be reused. In general, each of them *encapsulates* the control data and control functions. Encapsulation allows us to separate the purpose of an operation (function) from its implementation. In other words, we can focus on what the operations do instead of on how to implement them. Providing control algorithms with virtual operations (with virtual HGS's) enables us to set up actual links dynamically during run time. This makes it possible to construct dynamically modifiable control circuits. If a virtual operation is made pure, we obtain an incomplete specification, which essentially simplifies the testing and debugging of sophisticated algorithms.

References

[1] G. De Micheli, *Synthesis and Optimization of Digital Circuits*. New York: McGraw-Hill, 1994.
[2] S. Baranov, *Logic Synthesis for Control Automata*. Norwell, MA, Kluwer, 1994.
[3] S. Edwards, L. Lavagno, E. A. Lee, and A. Sangiovanni-Vincentelli, “Design of embedded systems: Formal models, validation and synthesis,” *Proc. IEEE*, vol. 85, pp. 366–390, Mar. 1997.
[4] G. Booch, *Object-Oriented Analysis and Design*, 2nd ed. New York: Benjamin, 1994.
[5] V. Sklyarov “Hierarchical graph-schemes,” Latvian Academy of Science, *Automatics and Comput.*, no. 2, pp. 82–87, 1984.
[6] ——, *Synthesis of Finite State Machines Based on Matrix LSI*. Minsk, Belarus: Science and Technique, 1984.
[7] V. Sklyarov, A. Adrego da Rocha, “Sintese de unidades de controlo descritas por grafos dum esquema hierarquicos,” *Electrón. Telecomun.*, vol. 1, no. 6, pp. 577–588, 1996.
[8] J. E. Vuillemin, P. Bertin, D. Roncin, M. Shand, H. H. Touati, and P. Boucard, “Programmable active memories: Reconfigurable systems come of age,” *IEEE Trans. VLSI Syst.*, vol. 4, pp. 56–69, Mar. 1996.
[9] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, and M. Trakhtenbrot, “Statemate: A working environment for the development of complex reactive systems,” *IEEE Trans. Software Eng.*, vol. 16, pp. 403–414, Apr. 1990.
[10] B. W. Kernighan and D. M. Ritchie, *The C Programming Language*. Englewood Cliffs, NJ: Prentice-Hall, 1988.
[11] R. C. Martin, *Designing Object-Oriented C^{++} Applications Using the Booch Method*. Englewood Cliffs, NJ: Prentice-Hall, 1995.



Valery Sklyarov received the Engineering degree from the Technical University—UPI, Uljanovsk, Russia, in 1972, the Ph.D. degree in computer science from the Technical University—BSUIR, Minsk, Belarus, in 1978, and the Doctor of Science degree in computer science from the Technical University—LETI, St. Petersburg, Russia, in 1986.

From 1972 to 1978, he was with the Research Institute, Minsk, Belarus, where he became Project Leader of the Design Peripheral Devices Group. From 1978 to 1994, he was with the Belorussian State University of Informatics and Radioelectronics (formerly the Minsk Radioengineering Institute), Belarus, as an Associate Professor, and from 1987 onwards, as Full Professor and the Head of the Computer Science Department. He is currently a Full Professor of computer engineering in the Department of Electronics and Telecommunications, University of Aveiro, Aveiro, Portugal. He has also been teaching and researching at Bialystok University, Poland, and Kassel University, Germany. He has authored and co-authored 17 books on subjects which include finite-state machine theory, computer-aided design (CAD), computer architecture, operating systems, and programming. His research interests include finite-state machine theory and object-oriented programming, with particular emphasis on their application to problems in logic synthesis and optimization.