Specifying Reconfigurable Control Flow for Open Architecture Controllers *

Chito Shiu, Michael J. Washburn, Shige Wang, Chinya. V. Ravishankar,
Kang G. Shin
Department of Electrical Engineering & Computer Science
The University of Michigan
Ann Arbor, MI48109-2122, USA

ABSTRACT

Reconfiguration is an important consideration in the design of open architecture controllers, so reconfiguring control flow is a critical design issue. This paper describes a specification method based on Finite State Machines that allows such reconfiguration. Our approach also provides a standard communication channel for cooperating controllers to exchange information and issue requests to each other. Our approach is being implemented and used for the testbed in the University of Michigan's open architecture controller (UMOAC) project at the NSF Engineering Research Center on Reconfigurable Machining Systems.
Key Words: open architecture controller, control flow, finite state machine

INTRODUCTION

Manufacturing equipment reconfiguration due to change in needs or technological obsolescence are common in modern manufacturing industry. The need for open architecture controllers (OAC) to accommodate such changes has been recognized since the early eighties (GM,1996)(Wright,1995). An open architecture allows flexible factory automation systems, rapid part realization, quality assurance, and rapid prototyping (Owen,1995)(Wright,1995).

Software modularization issues are integral to the construction of open architecture controllers since they must provide flexibility, ease of integration, and expansion. We need a sound architectural model comprised of blocks that are easy to integrate into useful applications. This goal requires the definition of a standard application program interface (API) for these component blocks so that related blocks can communicate easily and correctly. A properly designed public API provides a comprehensive interface that enables users to add or swap modules. UMOAC has chosen the object-oriented (OO) paradigm and C++ as tools towards this goal. Building blocks are formed as abstractions of physical objects (such as axes), manufacturing models (such as interpolators for axes), and utility modules (as for control flow or communication service). OO/C++ technology is chosen because it provides a way to separate the API from the implementation, and provides inheritance and polymorphism to make code more reusable. A component block is represented as a class in C++, with each class providing the services its corresponding abstract model is supposed to provide. For example, the clamp class object provides services that include clamping on and clamping off.

Abstract control flow specification is crucial to such a framework. A good control flow specification allows users to specify the desired controller behavior, while simplifying modifications when reconfiguration is necessary. The current commercial practice in providing a reconfigurable control flow specification is to use IEC-1131-3 (IEC,1993). Companies following this approach include Steeplechase, Nematron, and Wizdom. IEC-1131-3 introduced Sequential Function Chart (SFC), a supervision language, to specify control flow. A translator translates the machining program in IEC-1131-3 into C code (or some other computer programming language). The code is embedded into the soft-controller written in the same language and compiled into an executable that can run on the controller computer (Wisnosky,1996)(Steeplechase,1997). Provided that the control module API is made public, end users may change the machine control flow as follows: Write another IEC- 1131-3 program specifying control flow,

*The work reported in this paper was supported in part by the NSF ERC on Reconfigurable Machining Systems at The University of Michigan

use the translator to translate the program, put the translated computer code into the control module with the open API, and recompile the controller code. However, this solution for reconfigurable control flow does not support communication between controllers in different address spaces. This is a critical issue, since manufacturing applications have become increasingly complex, and cooperation between controllers is often required.

In this paper we will describe the effort of UMOAC to provide a reconfigurable control flow specification that also enables communication among the controller in different address spaces. In Section , we will justify our choice of the FSM model, and explain how event-based systems, more specifically machine tool systems, are mapped into FSMs. Section will explain how the representation can be simplified by using the hierarchical property of FSM. Section and will provide details of our approach to FSM implementation and reconfiguration. Section describes how the UMOAC uses the FSM approach, and Section concludes the paper.

EVENT-BASED SYSTEMS

Reactive systems are event-driven systems which react to external and internal stimuli (Boussinot,1991)(Harel,1987). In deterministic reactive systems, the order and values of inputs will completely dictate the order and values of the system responses. Therefore, system design and integration requires a specification method to describe the desired machining system behavior. We have chosen to use the finite state machine model to provide the specification.

FSMs for Control Flow

The FSM paradigm has been used for modeling various applications of reactive systems, such as human-machine interfaces and in the embedded controller domain (Kuuluvainen,1991)(Chiodo,1994). Although FSMs were mainly used in past applications to model hardware or embedded systems, they work well for our soft controller's needs. First, FSMs are an intuitive way to describe event-driven systems. In the Mealy machine model for FSMs (Hopcroft,1979), a transition works as follows: if the machine is in some state A and observes/receives event B, the machine will take specified action C and move to specified state A'. We can describe the requirements of a reactive system in the same way. Consider the following machine tool domain example: if the machine tool is in *ready* mode and receives an *axis feed* command, the axis performs the *feed* action, and the system state changes to *running.* Second, the standard FSM approach of sending and receiving events has been shown to allow concurrency and communication (Harel,1987). Such properties enable us to add reconfigurability and communication flexibility to our control flow specification. Third, this approach will allow us to perform model verification using the verifiable properties of FSM, such as reachability of states.

Machine Tool Modeling with FSMs

Users can describe desired machine tool behavior using FSM-based specifications. For example, one desired behavior may be to stop the machine tool if the tool status is *broken*. Each machining tool/station is first associated with a set of specifications, including tool capabilities (axis numbers and types) and environment stimuli accepted (axis position, tool status, temperature status, GUI inputs). In modeling machine behavior as an FSM, we model machine operations as the set of FSM actions, and the environmental stimuli as the set of FSM events.

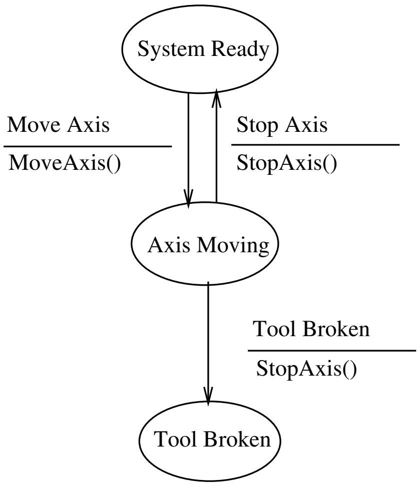

Fig. 1. FSM for a simple machine tool

The State Transition Diagram (STD) is a graphical method to describe FSMs. It is a directed graph, in which the vertices of the graph correspond to the states of the FSM. An arc labeled e/a from state q to state p in the STD represents the transition $q \times e \rightarrow p$, and triggers the action a. Fig. 1 shows a simple machine tool modeled as an FSM using STD, and has the capability to move an axis. The axis is either moving or stopped. There are three events in the system: two for the user's requests from the Graphical User Interface (GUI) to move or stop the axis, and one for the detection of a broken tool. This FSM describes a system that follows users requests for an axis, and stops the axis when a broken tool is detected. While this is a very simple machine, a more complicated system can also be modeled in the same manner. Nonetheless, because we want to provide a specification that is both comprehensive and well-modularized, we have decided to use multiple FSMs to represent complex machine tool systems. The issues of communicating and hierarchical FSMs will be discussed in depth in the following sections.

SYSTEM BEHAVIOR CONTROL WITH FSMS

A complex or integrated system may consist of several subsystems with different functionalities, and driven by different control flows. Some subsystems may have been developed separately, or already exist. To describe and control the behavior of such a complex system, as well as ease in the integration of new functionalities, we propose a hierarchical structure for FSMs to manage the system control flow at different levels.

Hierarchical FSM Organization

A complex machining system can be modularized as several coordinated subsystems, each of which possesses some desired functionality. The behavior of such systems can be decomposed into the behavior of the subsystems, whose control flows are described using separate FSMs. These FSMs are organized in a hierarchical structure, as shown in Fig. 2.

In Fig. 2, the Level-0 FSM is a top-level FSM, and corresponds to top-level system control flow. When the top level FSM reaches State 2 in Level-0, a Level-1 FSM is invoked and the Level-0 FSM delegates the control of the system to it. The Level-0 FSM remains at its current state until a new event arrives

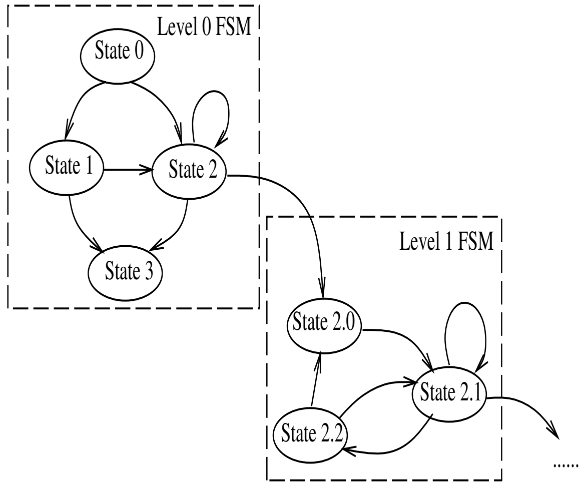

Fig. 2. FSM hierarchy

and causes a state change. Similarly, when the Level-1 FSM reaches State 2.1, it invokes a Level-2 FSM that controls a lower level subsystem, delegates control to it, and so on.

Although every FSM may be decomposed into several lower-level FSMs with one coordinating FSM, such a division should uphold the integrity of system components. That is, an FSM for a basic functionality should not be decomposed further. Only related or cooperating FSMs should be integrated under one higher-level FSM.

Under this scheme, each system component can be designed, developed, and tested separately, and then combined into a more complex application using a higher-level FSM to describe and manage the coordination among different modules and subsystems. This FSM hierarchy also helps isolate the subsystems and control flow changes. A control flow change for a particular component/subsystem only affects FSMs at that level.

The FSM hierarchical organization also aids in correctness verification. For a complex application system, it is natural to expect hundreds of states in the system, and even more transitions. It is difficult to verify the correctness of such a large system by finding isolated states (states without an incoming edge but that are not initial states), or dead states (states without a outgoing edge but that is not a terminating state). Instead of verifying a single FSM with a large number of states, we can decompose it into several FSM levels and verify them separately. The correctness of the control flow for the whole system can be verified by the correctness of each FSM node in the FSM hierarchy.

Communication Among FSMs

A fundamental issue in hierarchical FSMs is the communication between FSMs at different levels. Since the FSM is an even-driven mechanism, the next-level FSM invocation and termination are triggered by messages passing from a higher-level FSM to a lower-level FSM, or vice versa. For FSM objects at the same level, no communication is required, and the coordination of different FSM objects at the same level is handled by the higher level FSM. Only one top level FSM exists in the machine tool system.

The highest level FSM controls the execution of all the other modules. Generally, the Machine Control Logic FSM manages the control flow of all the other components by sending events to its lower component FSMs and gathering the current status of lower level FSMs if necessary. The Machine Control Logic FSM can be embedded in the same execution module as the highest level FSM, and invoked when the highest level FSM reaches the appropriate state. It can also be in another execution module, and actuated when that module reaches a state under the control of the highest level FSM.

The number of FSMs required depends on the complexity of the machining system and the reconfiguration requirements. For a simple machine (e.g., with only one axis moving from one fixed point to another one) and minimum reconfiguration (e.g., the operation mode won't be changed), a single FSM may control the entire system. But for a more complex system, every component may need an FSM to manage control flow, and several execution modules may be necessary. In this case, we need a highest level FSM to coordinate control flows among FSMs, and the Machine Control Logic FSM should begin to control the components when the top level FSM reaches a given state.

To support the communication between FSMs, each FSM module has an associated communication port. An FSM can only receive events from its own communication port. The communication port could be implemented using any mechanism the underlying system supports, e.g., shared memory, message queue, bus or network. Although different FSMs could share one communication port, only one FSM can consume a particular event and make a state change due to the event.

To invoke a next-level FSM, the higher-level FSM sends a message or event to the port of the next level FSM. Then, when the lower-level FSM is at a terminating state, a message or event is sent to its upper-level FSM. The communication between FSMs is shown in Fig. 3.

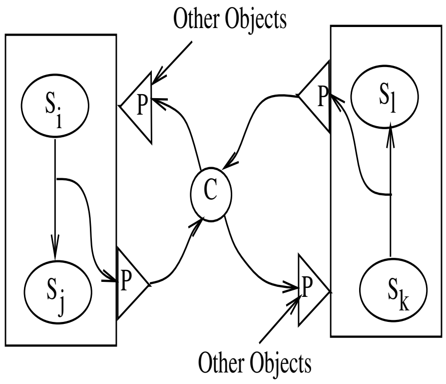

Fig. 3. Communication between different FSMs

In Fig. 3, when the Level-0 FSM makes the transition from S_i to S_j, the transition dispatches an event to an output port. This event is sent to the Level-1 FSM input port via a communication mechanism C. Upon receiving this event, the Level-1 FSM makes its state change and continuously receives events from its input port. Once the Level-1 FSM receives the event that makes it terminate (state change from S_k to S_l), it sends an event via C to notify the Level-0 FSM.

FSM IMPLEMENTATION

The additional requirements placed on our implementation of FSMs were efficiency and ease of use. Efficiency is important because FSMs will be used in applications that require real-time performance, such as machine control. To make FSMs useful in a reconfigurable environment, it must be easy for a user to integrate them into a control system without much modification.

The FSM is implemented as a set, or list, of states. Each state in this set contains an identifier as well as a pointer to a set of transitions. There is a separate transition set associated with each state (although two states could share a set if they needed the same transitions). Each transition contains an event number that triggers the transition, an action identifier that translates to the function associated with the transition, and a pointer to the next state (which will be in the original state set). Fig. 4 shows these structures.

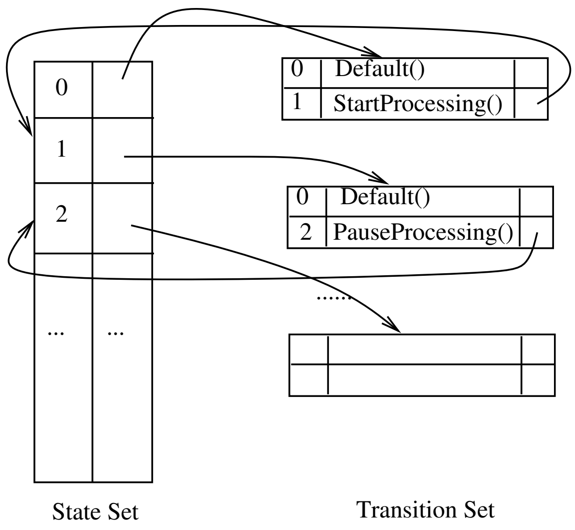

Fig. 4. FSM engin

State Transition Tables and Action Sets

The driver described in Section is identical for all FSMs, but the state transition table and the action set are different with the FSM. The state transition table specifies the contents of the state and transition sets, and is a computer-readable translation of the state transition diagram. The state transition table is stored separately in a file or database, and is loaded at system startup, configuration, or initialization.

The only code that needs to be written for a new FSM is that for the action set. The action set is the complete set of functions that may be called at state transitions, and contains all functions visible outside the module. Currently, the action set is implemented as a simple switch statement. This is a simple way to match the action number with its corresponding function.

FSM-BASED RECONFIGURATION

Machine system reconfiguration is necessary when there are changes in system components, functionality, or processing sequences. Each case requires change of control flow. Using the FSM mechanism, the control flow can be changed separately from the machine. Also, the FSM provides a method for the system designer/developer to specify application-specific control flow. Effort and skill required for reconfiguration can be reduced significantly with this approach.

Machine System Reconfiguration

Machine system reconfiguration can be classified as machine reorganization or processing sequence changes. In an OAC system, a machine system is modularized as different components, each of which accomplishes some designed functionality. Machine reorganization means using new or different components, or putting the components together differently for the same or different purposes. Examples include changing from a drill to a cutter, or breaking a 3-axis axis group into one 2-axis axis group and a 1-axis axis group. Processing changes are the changes in the operation sequence, while using the same machine with the same set of components. There are also many cases in which changes of components and processing sequences occur together. For real applications, the most frequent changes are processing procedures and/or a subset of components. It is seldom that the whole system or machine is redesigned or rebuilt.

Since both functionality and processing sequence are modeled by control flow in an OAC system, the control of a reconfigured machine system could easily be modified by changing the control flow of the system. In our FSM model, machine reorganization cause changes in the action set, where all possible invoked functions are defined and implemented. Processing sequence changes cause changes in the state table.

Mechanics of Reconfiguration

From the perspective of the FSM, reconfiguring a system with new control flow and components means changes of the states and transitions. As discussed in the previous section, such information is described in a system-independent state table and an action set. Thus, the reconfiguration of the system implies rebuilding of the FSM state table and action set. The procedure for reconfiguration with FSMs is shown in Fig. 5.

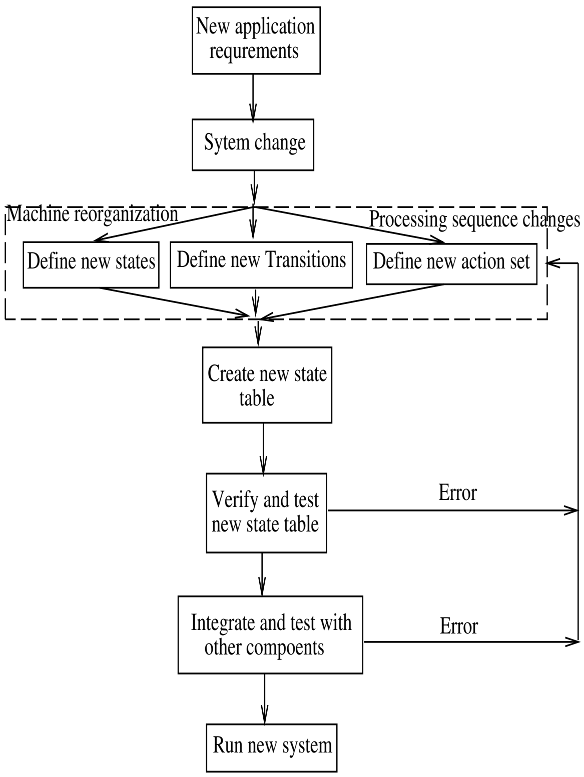

Fig. 5. Procedure of Reconfiguration with FSM

Changing in State Transition Tables and Action Sets

As Fig. 5 shows, the creation of a new state table is a major reconfiguration step. Since the FSM is organized as a hierarchy to control different components/subsystems, only those state tables related to the changes need to be rebuilt. For example, in a 3-axis milling machine system, FSMs may be used to control a 2-axis axisgroup, a 1-axis axisgroup and each axis. Creating a 3-axis axisgroup will introduce a new FSM to control the 3-axis axisgroup, but the FSMs for each axis will remain unchanged. Before the state table is created, its correctness should be verified. Since changes to a state table may introduce new events, states, and transitions, the FSM may be incorrect after such changes.

Action set changes are also necessary when the functionality of a component is changed. These changes are not as simple as those in the state table. The code for the FSM will need to be altered to call the correct procedures corresponding to the new functionality of the component. However, changes in component fuctionality mean that source code is being altered anyway, so the extra changes are not a great burden.

USE OF FSMS IN UMOAC

We illustrate how to use FSMs in a controller, by describing the structure of the Open Architecture Controller (OAC) Project software currently being developed at the University of Michigan.

Description of OAC Structure

The highest-level structure of the controller is Task. A Task represents an independent process (computational thread) in the system with certain properties, such as priority and period (if it is periodic). Every task contains a set of modules that it controls. A simplifed example from the UMOAC project appears in Fig. 6, and shows the use of three tasks.

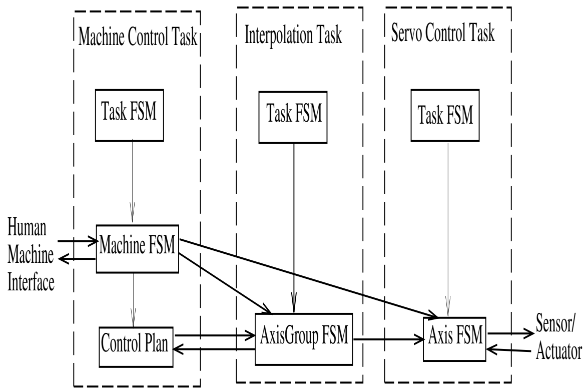

Fig. 6. OAC Task Structure

These tasks contain functionality vital to basic machine control. The Servo Control Task contains the functionality to run the servo controller (such as a PID control algorithm), and handles all of the I/O with the machine tool (encoders, tachometers, actuators). The Interpolation Task divides movements into the steps that single axis servo-algorithms can understand. The Machine Control Task handles all of the upper-level control relating to a particular machine. Its responsibilities include handling all data from the Human-machine Interface, sending configuration changes to the interpolation and servo control tasks, and ordering the motion commands for the machine.

FSMs and Controller Structure

FSMs are used to handle the control flow of each task. It would have been possible to create a large FSM that would control each task, but it made sense to use FSMs that controlled modules of smaller granularity. First of all, each task contains a Task FSM which can be reused for any Task object. One of these is the highest-level FSM in the system, controlling the configuration and startup of the other task FSMs. Each task FSM is responsible for the configuration and startup of each of its components.

Each of the three tasks shown includes a high-level module containing the functionality specific to that task (say, for Machine, AxisGroup, and Axis). Each of these is modeled by an FSM that can be considered below the task FSM in the hierarchy. The Task FSM includes an execution state that allows the FSM directly below it in the hierarchy to run. This is shown in Fig. 7, which is a breakdown of the Machine Control Task hierarchy.

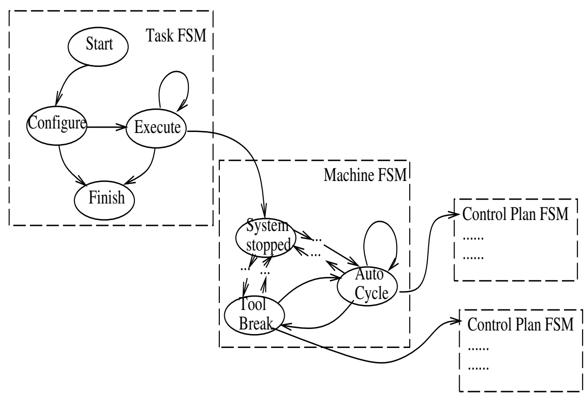

Fig. 7. FSM Hierarchy of Machine Control Task

The Machine Control Task contains another level FSM that controls the part program execution. The machine may run a different part program depending on what state it is in. The hierarchical property of FSMs makes this easy to do, because each state of the machine FSM can have a part program FSM subset, as Fig. 7 shows.

In this FSM hierarchy, the Machine FSM controls all the other components of the sytem once they are allowed to execute by their respective task FSMs. Once the tasks are put into the executing state, the Machine FSM controls the execution of the entire system, unless an exceptional condition or a shutdown occurs causing the Task FSM to change state. Also, the Part Program FSM controls the AxisGroup FSM, which controls the Axis FSMs.

CONCLUSION

We have shown how to design and implement control flow specifications using the FSM model. Users can reconfigure the controller using this paradigm. The FSM representation allows both hierarchical design and communications between FSMs. The hierarchical design allows us to decompose a complex system effectively, to ease the reconfiguration and verification process.

The goal of Open Architecture Controllers is to allow components to be added to, and removed from, the system without major disruption. The FSM approach helps allows control flow for a new component to be easily integrated with the existing pieces. The approach also allows easy modifications to the control flow of the existing components, by simply modifying the associated state table. This has been a very useful property, as the project has gone through many iterations, and in some cases, dramatic changes in the methods used for control. Of course, if the functionality is altered, changes will go beyond the state table (Section 5.3), but the FSM still gives the property of localizing the effect of modifications.

Acknowledgements

The authors would like to thank Sushil Birla of General Motors, as well as Professors Yoram Koren and Galip Ulsoy of The University of Michigan, for their input and valuable discussions.

REFERENCES

GM Powertrain Group Manufacturing Engineering Controls Council, 1996, *Open, Modular Architecture Controls at GM Powertrain — Technology and Implementation, version 1.0,*.

Wright, P. K., 1995, "Principles of open-architecture manufacturing," *Journal of Manufacturing Systems,* pp. 187-202.

Owen, J. V., 1995, "OPening up controls architecture," *Manufacturing Engineering,*, pp.53-60.

Wisnosky, D.E., 1995, *SoftLogic: Overcoming Funnel Vision,* Wizdom Controls, Inc.

Steeplechase Software, 1997, *Steeplechase Handbook, A Practical Guide to PC-based Control & Flow Chart Programming, 1st Edition,*.

Kuuluvainen, I., Vanttinen, M., Koskinen, P., 1991, "The action — state diagram: A compact finite state machine representation for user interfaces and small embedded reactive systems," *IEEE Transactions on Consumer Electronics, Vol. 37, No. 3,* pp. 651-658.

Chiodo, M., Glusto, P., Jurecska, A., Hsieh, H., Vincentelli, A., Lavagno, L., 1994, "Hardware-software codesign of embedded systems," *IEEE Micro,* pp.26-35.

Harel, D., 1987, "Statecharts: Avisual formalism for complex systems," *Science of Computer Programming, Vol. 8,* pp. 231-274.

IEC, 1993, *Programmable Controllers Part 3: Programming Languages,* International Standard.

Boussinot,E., De Simone, R., 1991, "The esterel language," *Proceedings of the IEEE, Vol. 79, No. 9,* pp. 1293-1303.

Hopcroft, J., Ullman, J., 1979, "Intoduction to Automata Theory, Languages, and Computation," Reading, Mass. : Addison-Wesley.