Proceedings of the American Control Conference
Anchorage, AK May 8-10, 2002

Reconfigurable Logic Control using Modular FSMs: Design, Verification, Implementation, and Integrated Error Handling[1]

S. S. Shah
The University of California
Department of Mechanical Engineering
Berkeley, CA 94720
sshah@newton.berkeley.edu

E. W. Endsley, M. R. Lucas, and D. M. Tilbury
The University of Michigan
Department of Mechanical Engineering
Ann Arbor, MI 48109-2125
{ericend,mrlucas,tilbury}@umich.edu

Abstract

This paper describes the design and implementation of logic controllers on a small-scale machining line testbed using modular finite state machines. The logic is verified to be internally correct before being implemented on the testbed. Reconfiguration of the controller for a new manufacturing scenario is demonstrated, as is the integration of error handling. The ease of use of this modular finite state machine design methodology is discussed, as is the complexity of the resulting designs. Algorithms are presented for design, reconfiguration, and error handling integration.

1 Introduction

Reconfigurable machining lines are manufacturing systems that eliminate the need to create a new physical system when changes in the final product are made [4]. With reconfigurable machining lines, rapid redesign of a system is made possible by making changes to an older system. In addition to the mechanical reconfiguration of the line, the logic controllers for the system must be reconfigured.

In current practice, the control algorithms for manufacturing systems are written in ladder logic and implemented on proprietary computers called programmable logic controllers (PLC). The logic is extremely complex and a change in the manufacturing scenario can involve rewriting the entire code. It is commonly noted by industrial practitioners that "I generally find it easier to start over from scratch rather than modify an existing [Ladder Diagram], including my own." [7, p. 110]

One logic control language that has been proposed for manufacturing systems is based on Finite State Machines (FSMs) [1]. In the Modular Finite State Machine (MFSM) design methodology, there are control modules for each machine module in addition to coordination modules. The interfaces between the modules are well-defined, and used for all communication. Theory exists so the combination of the modules are guaranteed to be internally correct (no deadlocks). This methodology restricts definition of the manufacturing scenario (operation sequence) to only one control module, the control plan. The other control modules are independent of the operation sequence. Thus, when the manufacturing scenario is altered, only the control plan needs to be changed. Also, if mechanical modules are added to or removed from the system, the appropriate mechanical control modules must also be added or removed. The modularity of the control structure parallels that of the mechanical structure, and enables the reconfigurability of both.

In sharp contrast to ladder programs used in industry, MFSM programs exhibit a strong modularity, a lack of global variables, and are verified at the design stage. These departures from the current programming practice raise questions of ease of use, implementation, reconfiguration, and extension of MFSM logic programs to which we formulate preliminary answers. First, we define algorithms for implementing and reconfiguring MFSM logic controllers. Second, we follow these algorithms to implement and reconfigure a logic controller. Finally, we evaluate the resulting programming process and code. The results of this research can be used to compare the time needed not only for the programming and debugging of the code to run the system, but also for the reconfiguration of a manufacturing scenario.

The outline of this paper is as follows. Section 2 gives more background on logic control for manufacturing systems, and describes our testbed. Section 3 provides an algorithm for developing logic control using MFSMs and section 4 for reconfiguring an existing system. Section 5 outlines the method for including error handling in the logic control. Finally, section 6 presents conclusions and suggestions for future work in using MFSMs.

2 Background and testbed

Manufacturing systems consist of many machines working together that require many inputs and outputs. Also, the machines must be coordinated to work simultaneously and be able to handle error conditions. These factors combine to make the logic necessary for manufacturing systems extremely complex.

2.1 Logic control issues

Logic control has historically been programmed in relay ladder logic, a low-level programming language, and the programs for even a relatively small system rapidly become unwieldy. Although industry has recently moved towards developing one logic controller for each station on a line, each logic program is still extremely complex.

Even though the functions of logic controllers for different machines may be similar, programming the control logic

[1]This research was supported in part by the NSF under grant EEC95-92125.

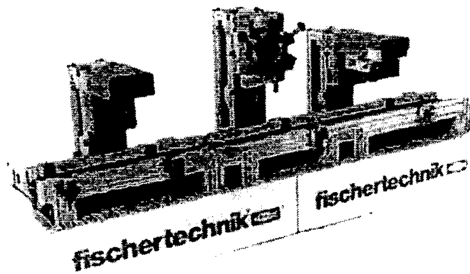

Figure 1: The manufacturing line testbed.

takes approximately 50% of the total construction time for each new system since there is not yet a standard integrated tool with which to carry out formal analysis of correctness. Validation through simulation is starting to be used, but many control systems are first tested on the mechanical system. A relatively long "cycle and debug" stage in the development process is needed. The reduction in product life-cycles has driven an economic need for shorter design and debug phases for the control code. Functional models of the machines are not typically constructed, so there is no reference model against which the control logic can be compared. As a result, it is not possible with current industry practice to conduct formal analysis and systematic design of logic controllers.

2.2 Manufacturing line testbed

The experimental testbed used in this work has three workstations: a drill unit, a vertical boring machine with a three-position tool changing turret head, and a horizontal milling machine as shown in Figure 1. A conveyor belt is used to transport the work pieces.

The testbed has 15 inputs (limit switches and proximity sensors) and 12 outputs (motors). All I/O is binary (on/off). The system interfaces with a PC through a digital I/O card. The software used to implement the control detects changes in the input signals as events; rising edges are distinguished from falling edges.

2.3 Specifications for logic control

Two manufacturing scenarios were considered. In the original scenario, the drill drills one hole and the horizontal mill does one pass; the vertical mill is idle [2]. The sequence of events needed to complete this scenario are shown graphically in a timing bar chart in Figure 2. Operations that are causally linked (one should be started when the other completes) are indicated by vertical lines. In the revised scenario, the drill drills four holes, the vertical mill uses all three tools, and the horizontal mill does two passes [5]. Figure 3 shows the timing bar chart associated with this scenario. Short moves of various motors are used for a slight movement rather than a movement to a proximity sensor or limit switch. After implementing the overall control on both scenarios, error handling is added to the original scenario.

2.4 MFSM software

An FSM represents a discrete event system. FSMs are visualized by a set of nodes representing states connected by arrows representing transitions. A transition from one state

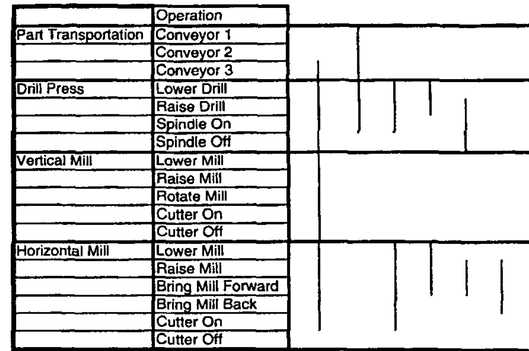

Figure 2: The timing bar chart for the first scenario.

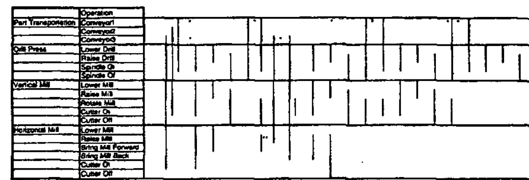

Figure 3: Second scenario timing bar chart.

to another is triggered by an event which labels the arrow. A transition may also have actions associated with it, denoted separately from the event with a slash symbol. When the event signaling a transition occurs, the actions are performed and the system moves to the new state.

To implement the MFSM controllers on the testbed, software developed at the University of Michigan was used [3]. Each control module is represented as a finite state machine (FSM). The MFSM Software is used to verify that each module checks to the ports connected to it. In addition, when modules are combined, the software determines whether the communication between modules occurs correctly. If the communication is incorrect, a transition to the dump state is created, highlighting the location of the error. The verification software is written in Java and can create a single FSM from the collection of modules, which can then be converted into a set of "switch" statements. Headers and footers are added to these switch statements to create a complete C program for implementation.

3 Developing logic control

In this section, we provide an algorithm to show how MFSM methodology can be used to develop logic controllers for manufacturing systems, and then illustrate this algorithm by applying it to the testbed example.

Algorithm 1 *Logic Control Construction*

1. *Determine the modular structure of the system, with one module for each workstation as in Figure 7.*
2. *Determine the coordination structure between workstations. Either a centralized control module or multiple coordination modules (one per workstation and additional header and footer modules) can be used.*
3. *Define the coordination control module(s).*

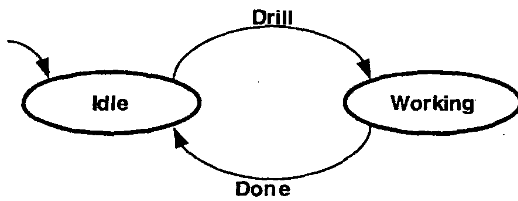

Figure 4: Port between drill controller and drill control plan.

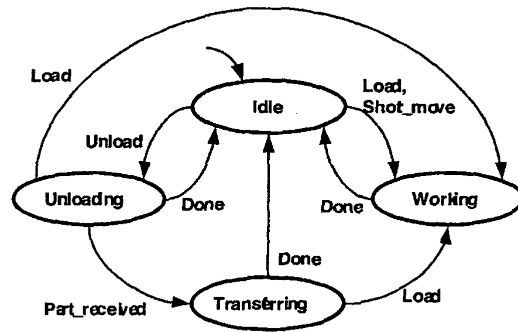

Figure 5: Port between conveyor and workstation control plan.

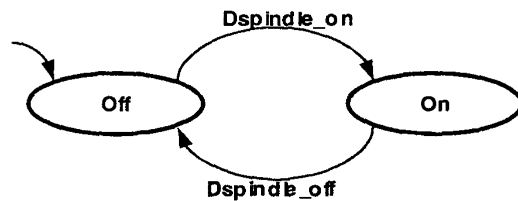

Figure 6: Port between drill controller and spindle motor.

   (a) *Write the coordination control module(s). If multiple coordination modules are used, each coordination control module should communicate with a workstation (through its control plan) and two other coordination control modules. It must communicate when to start the next station or stop the previous station to coordinate transfer of parts. A header and footer coordination control module should be defined for modularity.*
   (b) *Define the ports in the control structure.*
   (c) *Covert the modules and ports to the FSM text format needed for verification.*

4. *For each workstation,*
   (a) *Determine the hierarchical modular structure as seen in Figure 8. This structure should include a control plan (the sequence of operations) and mechanical control modules which interface with the physical I/O. Each I/O point will communicate with only one mechanical control module.*
   (b) *Write the control plan module. A sample control plan is shown in Figure 9.*
   (c) *Write the remaining mechanical control plan modules.*
   (d) *Define the input and output ports for each mechanical control module. One port of each mechanical control module should interface to a higher-level control module and remaining ports with the I/O or lower-level control modules. Ports connecting control modules can be simple two state ports such as Figure 4 or more complex such as Figure 5 to disallow some communication. Ports to the physical system are generally simple as shown in Figure 6.*
   (e) *Convert all modules and ports to FSM text.*
   (f) *Use MFSM Software to verify every module to each of its ports. If the modules do not check, modify the control modules and ports defined in steps 4b–4d. Repeat steps 4e and 4f as needed.*
5. *Combine the control modules into a single FSM and verify the final module has no states connected to the dump state. If the verification fails, again modify the control modules and ports defined in steps 3–4 and repeat as needed.*
6. *Convert the final combined FSM to C code and implement on the hardware.*

When the algorithm converges, the resulting logic controller is deadlock-free [3]. The verification of modules to ports and the error messages from the combinations will locate errors in the logic by pinpointing the module which needs to be redefined so that the resulting logic controller will be verified as correct. In contrast to current practice using ladder logic programming, this methodology requires a significant outlay of effort up-front to verify the absence of internal deadlocks. Although this may lengthen the programming process, it should decrease the time required to debug the code on the physical system, and increase code reusability.

Example 1 (Testbed control development)

Step 1. To implement the simple configuration (see Figure 2), we divided the system into three workstations.

Step 2. To move parts to the next workstation in the line, the conveyor of the current and next workstation must be on, so a control module on a higher level is needed. We began by attempting to define a single module, but this module needed over 50 states and 100 transitions and was too difficult to complete. Instead, conveyor coordinator modules were created for each workstation. These coordinators communicate with a workstation and two other coordinators to keep track of whether the next workstation is ready for a part, whether the previous workstation is sending it a part, and whether its workstation is working on a part.

Conveyor coordinator header and footer modules were added as well. The header and footer modules were designed to send the same commands and responses that a coordinator module does, but made to be first and last in the sequence. These modules allow the coordinator modules to be identical and therefore modular. See Figure 7 for overall control structure.

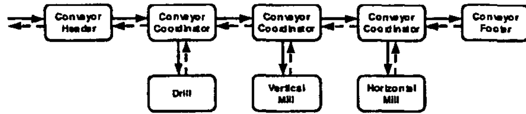

Figure 7: Overall control structure.

Step 3. The state diagrams for the coordination modules were then drawn. The complete logic controller for the testbed can be found in [6].

Step 4. We began writing the logic control for the drill. We grouped the motors for the drill into those needed for transport (conveyor motor) and those needed for drilling (slide and spindle motors). A control plan was defined to coordinates these actions. The diagram for the drill control plan is shown in Figure 9. The initial state, indicated by the short arrow, is **Idle**. When the **start (1.st)** command from port 1 arrives, a **load (2.load)** command is issued through port 2 and the state transitions to **Moving1**. The rest of the diagram can be interpreted in the same manner. The ports that this module uses can be seen in Figure 8; they are numbered starting counterclockwise on top: port 1 of the drill control plan corresponds to port A, 2 to B, and 3 to C. A drill controller was placed below the control plan that coordinates the spindle and slide. Because the spindle only needed to be turned on or off, its behavior was modeled by a port. The slide, however, needed to coordinate the slide motor with the limit switches so a slide module was created. Finally, a conveyor module was placed beneath the control plan to coordinate the conveyor motors with the proximity sensors. The control hierarchy of the drill is shown in Figure 8. Finally, the ports between each module were defined. Each module was checked to its ports.

Next, we began writing the logic for each of the mills. The slide and conveyor modules used for the drill could be reused. The vertical mill does not use any of its tools in this scenario, so its logic was easy to complete. However, the motion of the horizontal mill was more complicated and we realized we needed a separate module for each degree of freedom (vertical and horizontal motion). The modules and ports for these workstation were converted to FSM text and the ports were verified to the modules.

Step 5. Next, we began combining the modules into a single module representing the entire system. If a combination led to a state being connected to the dump state, the FSM Verification software would give an error. We located the combination that gave the first error, displayed that module and fixed that problem by modifying the control module. We repeated this debugging process until the combination was error-free. The combined FSM had 1102 states.

Step 6. Finally, we ran the FSM to C conversion, resulting in 97893 lines of code (including comments and white space), and implemented it on the testbed.

4 Reconfiguration of logic control

One advantage of the MFSM design method over existing techniques is the ease of reconfigurability due to the modularity of the logic control. In this section, we provide an algorithm for the reconfiguration of an already existing system, and use the testbed as an example.

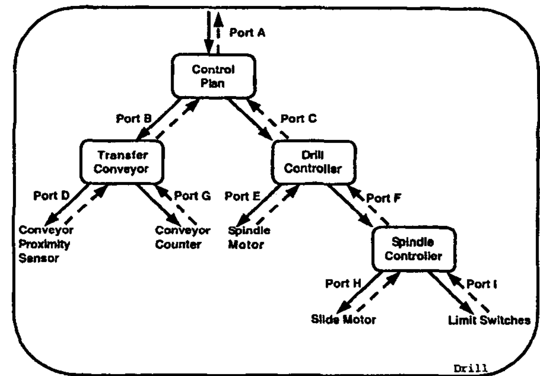

Figure 8: The control structure for the drill.

Algorithm 2 *Logic Control Reconfiguration*

1. *Determine what new modules are needed and place them in the existing control structure. New modules are needed when a previously unused motor or sensor is used. Define these modules and their associated ports in the same manner as in Algorithm 1.*
2. *Determine what modules (other than the control plan) and ports need to be changed and make the alterations. Sometimes, new commands in previously defined modules will be necessary if the new scenario requires new functionality.*
3. *Alter the control plan modules. The states associated with the loading and unloading of the part should be grouped together since theys will usually remain unchanged. The remaining states will be those associated with the operations the workstation performs and will need to be altered for the new scenario. Alter the ports if necessary.*

Example 2 (Testbed control reconfiguration)

Step 1. We reconfigured the original scenario to the one shown in Figure 3. All the modules needed for this scenario were already defined.

Step 2. However, since the drill and horizontal mill needed to use short moves of the part and slide (only the mill), new ports and states were added to the conveyor module and horizontal mill vertical motion controller module.

Step 3. Finally, the control plans were changed by grouping states as described in step 3 and selecting the states associated with working on the part for alterations. With the drill and horizontal mill, a new state was added in case the short move command moved the part off the workstation. See Figures 9–10 for the control plan of the drill workstation for both scenarios. In the reconfigured drill control plan, the port numbers are unchanged from the first scenario. The addition of the states **Drilling2-4** for the extra holes and **Moving3-5** for the short moves of the parts. Also, an **Off Part** state was added in the case where the short move moves the part off the workstation. In addition, two new states were added to the vertical mill so that the tool changer was in the home position before working

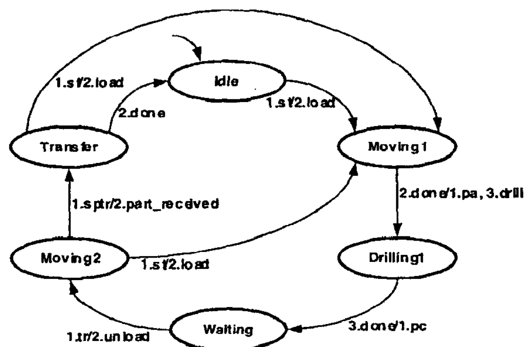

Figure 9: Drill control plan for original scenario.

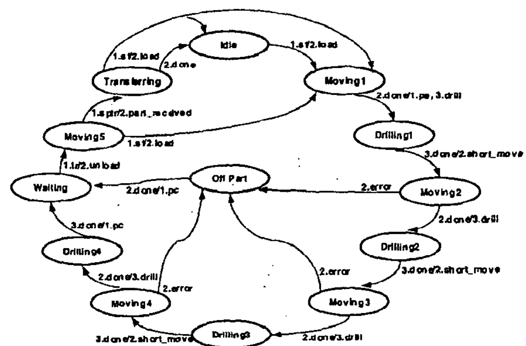

Figure 10: Drill control plan for reconfigured scenario.

on the first part. The transfer of the parts between the workstations remained the same so the conveyor header, coordinators, and footer were not altered. The complete logic control diagrams can be found in [6].

In the both scenarios, twenty modules and thirty-six ports were used. Of the twenty modules in the second scenario, fourteen were reused from the first with no alterations. Three of the six that were altered were the control plans for each workstation. Seven states and ten transitions were added to the drill control plan, eight states and eight transitions to the vertical mill, and four states and five transitions to the horizontal mill. The horizontal mill vertical motion controller module was altered to allow short moves by the addition of two states and six transitions. The final two altered modules were the conveyors for the drill and horizontal mill where the addition of two states and six transitions allowed for short moves of the part.

Finally, the new control modules were combined. Because most of the modules were identical to the previous scenario, any errors were associated with the altered modules or the new commands making the errors simple to locate. The combined module was then converted to C code and implemented on the testbed.

5 Integration of error handling in logic control

One important function of a logic controller is to handle errors and exceptions. However, the procedures described above do not explicitly take into account error handling. Possible errors in a logic control system can be divided into two categories: an unexpected event occurs or an expected event does not occur. We provide an algorithm to show how error handling can be added to a MFSM logic controller. We then apply this algorithm to our example for two different error cases in the original scenario: an unexpected part entering the middle of the machining line, and an expected part not completing its transfer to the next workstation.

Algorithm 3 *Logic control error handling*

1. *For either error case, identify the module lowest in the control hierarchy that needs to recognize the error. Appropriate commands and states for the error handling should be added to that module's state diagrams.*
2. *Move to the next level up in the control structure and make any necessary alterations. This step should be repeated until no further changes are needed, or the coordinator module is reached.*
3. *Alter the coordinator module as needed and use new coordinators for all workstations. Any changes in the coordinator module may propagate down the control hierarchy.*

Example 3 (Integration of unexpected part error)

Step 1. We began by allowing the conveyor module to identify a part if it is idle with no part and to alert the control plan.

Step 2. Because the operations in a system usually need to be performed in a specific order, this part must not be worked on. Therefore, the station at which it is placed must recognize the part as a "bad" part and send it to the next station. The control plan was recoded to enter a new state, turn its conveyor on, and let the coordinators know that an unexpected part is present.

Step 3. A new command was added to the coordinator so it can tell the next coordinator of the unexpected part. Subsequent workstation control plans were altered as in step 5 so the part is passed to the end of the machining line.

Example 4 (Integration of timeout error)

Step 1. We added a transfer timer to the conveyors that signaled an error if the it finished before a part was received.

Step 2. The control plans for each workstation were altered to return to idle if the conveyor sent out a timeout error. This meant the workstation assumed the part was lost and could continue working as normal with future parts.

Step 3. The coordinators for the workstations involved in the transfer were trasitioned from working to idle due to this error.

The coordination modules (including the header and footer) as seen in Figure 7, the control plans for each workstation, and the conveyor modules as seen in Figure 8 for the drill needed to be altered to handle both error cases. The new drill control plan can be seen in Figure 11. We added the response of **error** (when the timer runs out before a part is received) and **unexpected_part** (when a part is received

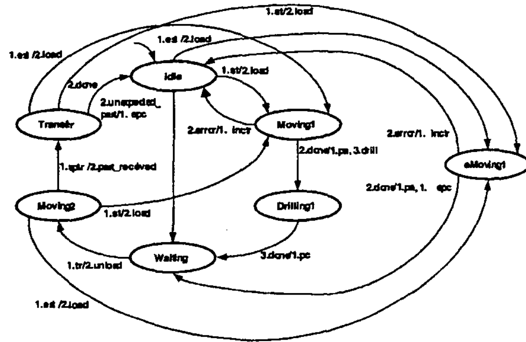

Figure 11: Drill control plan with error handling.

in the idle state) to the conveyor module. The control plan modules of each workstation were altered to respond with an **epc (error part complete)** to signal that the "bad" part was ready to be transferred. In addition, the response of **inctr (incomplete transfer)** was used to alert the coordinator the part sent was never received. Also, each control plan was allowed to accept the command of **est (error start)** to start its conveyor, but not work on the part. Finally, the conveyor coordinator modules were allowed to communicate whether a "bad" part had arrived or a part had never arrived at its station (by using **epa (error part arrived)** and **inctr (incomplete transfer)**). They also would tell the next coordinator to start its station in error mode (**est (error start)**). The complete logic, including both error handling scenarios, can be found in [6].

A total of eleven modules needed to be altered to include error handling. Three of these were conveyor modules and were changed identically by adding two transitions and altering one. One state and seven trasitions were added to each workstation control plan. Five transitions were added to the coordinator header, one state and twenty-one transitions were added to each of the three coordinator modules and one transition to the coordinator footer.

6 Conclusions and Future Work

The use of FSMs in the logic control of manufacturing systems allows the complexity of the manufacturing scenario to be placed in the control plans of each workstation so alterations to the scenario only involve changing the control plan. Additions, such as error handling, are relatively simple to make. After the first scenario was implemented on the testbed, the second scenario took only a few days to implement. The error handling was inserted in a few hours. This demonstrates the use of MFSMs makes these changes easy and fast.

In the future, We would like to reconfigure a machining line only by changing the only control plan (eliminating steps 1 and 2 in Algorithm 2). Developing code for different scenarios would create all possible commands in previously defined modules. The modules would reach completion and reconfiguration would be a single step.

Also, the more complicated second scenario led to an extremely large combined module (about 8500 states). The Java program ran out of memory when combining the modules, and when converting the combined module to switch statements. In addition, the C compiler ran out of memory. If the MFSM could be executed in a modular fashion, memory errors would be eliminated (although this would require altering the existing software environment).

The error handling that we worked on was specifically for two cases. More work on error handling should be done for broader cases. While testing the testbed codes, sensors were brushed accidentally and the system would enter the wrong states. It should be possible to modify the controller to handle these errors correctly, and the system would become much more robust.

References

[1] C. G. Cassandras and S. L. Lafortune. *Introduction to Discrete Event Systems.* Kluwer, Boston, 1999.

[2] A. Denault. Logic control reconfiguration for a machining line testbed. Technical report, University of Michigan, Mechanical Engineering, April 2001. ME 490 Independent Research Report.

[3] E. W. Endsley, M. R. Lucas, and D. M. Tilbury. Software tools for verification of modular FSM based logic control for use in reconfigurable machine tools. In *Proceedings of the Japan-U.S.A. Symposium on Flexible Automation*, 2000.

[4] Y. Koren, U. Heisel, F. Jovane, T. Moriwaki, G. Pritschow, G. Ulsoy, and H. Van Brussel. Reconfigurable manufacturing systems. *CIRP Annals—Manufacturing Technology*, 48(2):527–540, 1999.

[5] S. S. Shah. Logic control design for a machining line testbed using modular finite state machines. Technical report, University of Michigan, Mechanical Engineering, December 2000. ME 490 Independent Research Report.

[6] S. S. Shah. MFSM logic control programs for the machining line testbed, August 2001.

[7] V. VanDoren. Designing PLC-based control without ladder logic. *Control Engineering*, 43:110, June 1996.