Proceedings of the 2001 IEEE
International Conference on Robotics & Automation
Seoul, Korea · May 21-26, 2001

Modelling Manufacturing Control Software *

Orest Storoshchuk
McMaster University
email: orest.storoshchuk@gm.com

Shige Wang and Kang G. Shin
Real-Time Computing Lab,
EECS Department,
The University of Michigan
email: {wangsg,kgshin}@eecs.umich.edu

Abstract

Software is essential in today's machine controllers for reuse, reconfiguration and cost-reduction. Formal models and specifications for such a system are critical to guarantee correctness. In this paper, we present an approach to model manufacturing control systems based on a framework for subsystem composition and a Nested Finite State Machine (NFSM) model for system behavior. The contribution of this work is the application of model-based methodology for real control systems. Some preliminary results have shown that development costs are reduced significantly due to domain-specific knowledge captured in such a model.

Key Words: control systems, formal model, software architecture, real-time systems.

1 Introduction

Manufacturing control systems typically control a set of equipment with various real-time information pertaining to the processes and equipment. In current practice, control engineers first develop the code for supervisory discrete event control. Then, the data acquisition/monitoring code developed by engineers from different groups is added. Such a methodology results in ad hoc system implementation, with success highly dependent on individuals' efforts [1]. Adding the monitoring code requires substantial effort first to comprehend the control code and then to integrate without compromizing performance. Regular maintenance incurs more costs to keep monitoring synchronized with the continually evolving control code.

Representing domain knowledge and information within system context is essential to facilitate software development and maintenance. It has been shown that component-based system design and integration will help solve these critical issues [2, 3, 4]. In practice, however, how to apply these solutions and methodologies in the manufacturing domain still remains unanswered [5]. There is little evidence showing significant improvement in control software development [6, 7, 8]. One fundamental reason for this is inadequate scientific guidelines for adapting these methodologies in the control domain. Misapplication can result in an increase in effort, duration and skill requirements. Linkman and Rombach [8] outlined an experimentation and research cycle necessary to support the evolution of software engineering as a discipline. The experimental process is needed to successfully transfer software technologies from the theoretical domain into industrial applications.

This paper presents how to construct control software for a real system using a model-based approach. The construction demonstrates the first portion of research on the experimental process of transferring technologies to industrial applications and the evaluation of the model-based approach. We chose a stamping/roll-forming work-cell system as an application to model, which is representative in the manufacturing domain. The model that we used combines the object-oriented composition model and a formal method of finite state machine [9, 10], which is compliant with the OMAC user group specifications [11]. We focus on providing a process for assembling the components and their behaviors. In this process, the domain knowledge was modeled and represented as a set of reusable components, which can be selected and integrated by a non-controls user. It has been shown experimentally that the process removes the need for specialized controls engineering knowledge for control software design and implementation.

The rest of the paper is organized as follows. Section 2 describes the software architecture and the stamping/roll-forming experimental system. Section 3 presents the components, specifications and behaviors of the work-cell system. Section 4 presents how the construction of the experimental system is achieved, and

*The work reported in this paper was supported in part by DARPA under Grant F33615-00-1706 administrated by the US Airforce Wright Lab.

describes our experience on efforts measured during the development. The paper concludes with Section 5.

2 Software Architecture and Experimental System

2.1 Software architecture

Software for manufacturing control systems can be divided into two parts, controller software and application software, as shown in Figure 1. Controller software defines the functionality of the system and is platform-dependent. The application software defines the behavior of the system and is expected to be platform-independent.

In our architecture, the software of a control system is modeled as a set of inter-communicating components. Components are designed and implemented with a set of external and internal interfaces, as well as environmental protocols, as shown in Figure 2. Inside each component, there is a control logic driver devised to support separate control logic specifications. This is the mechanism that separates the behavioral information from the functionality of the component that can be implemented without the control domain knowledge. The environmental protocol enables the component to be used in various platform configurations. Further, the existence of internal/external event mapping enables the implementation of the component independently of its deployment. Therefore, components can be either purchased from various vendors, or ported from other existing applications, or specially developed for a particular application.

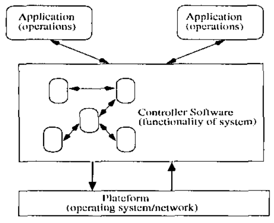
Application (operations)
Application (operations)
Controller Software (functionality of system)
Plateform (operating system/network)

Figure 1: Controller software structure

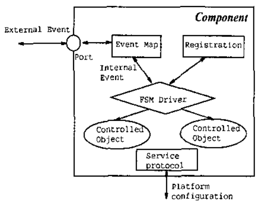

Figure 2: Software component structure

To support composition with different granularities, components are organized hierarchically. The behavior of a component can be specified independently of its implementation and the behavior of other components. The behavior of each basic component is modeled as a Finite State Machine (FSM), and the behavior of a composite component is modeled as a Nested Finite State Machine (NFSM) which synchronizes the FSMs of its member components. The FSM we are using is a Mealy machine. The existence of a control logic driver in each control-related component enables direct execution of such separate behavior specifications.

The NFSM behavior model has two significant advantages. First, it supports integration and verification of behavior in a hierarchy, which is essential when we model software with components. This reduces the complexity of system behavior analysis and verification. Furthermore, the properties of a system can be analyzed more thoroughly using a formal method instead of using costly simulations. Second, since a FSM can be fully specified in a table format, it is convenient for implementation and run-time reconfiguration of control logic. Moreover, such an implementation enables control engineers to utilize it in their daily routine, as it does not require specialized software tools (such as programming languages, compilers and debuggers).

2.2 Work-cell control system

To derive the process of software development with our model, we constructed a typical manufacturing system — the stamping/roll-forming work-cell shown in Figure 3 — and obtained the process by generalization from this specific case. This work-cell is representative of the higher end of complexity typically found in Roll Form manufacturing. The destacker descends onto a stack of metal sheets, grabs a single plate, then lifts and carries it over to the press entry conveyor. It then descends and deposits the sheet onto the conveyor, which delivers it into the press. The press stamps the plate, and then the press exit conveyor removes it to the scissors lift. The scissors lift lowers it onto the roll former entry conveyor, which delivers it into the roll former. The roll former shapes the sheet by forcing it through rolls, which have the contour of the desired finished part and then deposits the part on the finished part table.

3 Work-Cell Composition and Specification

To model the system with an object-oriented model, the work-cell is partitioned hierarchically. At the top level, the entire work-cell is treated as an independent system which could be operated by itself. The behavior of the overall system can be specified with a system-level FSM. At the next level, the system is divided into distinct subsystems as shown in Figure 4, coordinated by the system-level state machine. The behavior of each subsystem is specified with a subsystem-level FSM. A subsystem is then further divided into smaller components, which are coordinated by their subsystem-level state machines.

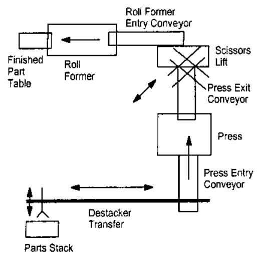

Figure 3: The work-cell system

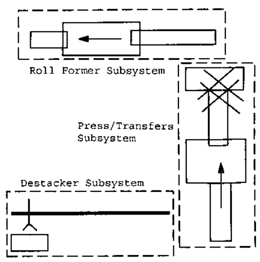

Figure 4: Partitioning of the work-cell system

The components at each level can be modeled as classes in an object-oriented model, and can be implemented as software components. The abstract classes and their nesting are shown in Figure 5. The top layer defines the system-level control that is responsible for overall functions and behaviors such as sequentially starting subsystems (to prevent overloading the electrical distribution system), determining when the entire cell can be placed into an automatic mode, running the cell automatically, emergency stopping, etc. A system-level state machine models this behavior. The next level is the subsystem control that groups closely related components together to provide coordination and ability of independent subsystem operations. Classical principles of modular software design, such as coupling and cohesion, can be used to determine the boundary and function of a subsystem. A subsystem-level state machine is responsible for subsystem behaviors such as sequentially starting components, setting the overall behavior mode (automatic, manual, off), single cycling, informing the system of the overall subsystem state, etc. The lowest level defines the individual components.

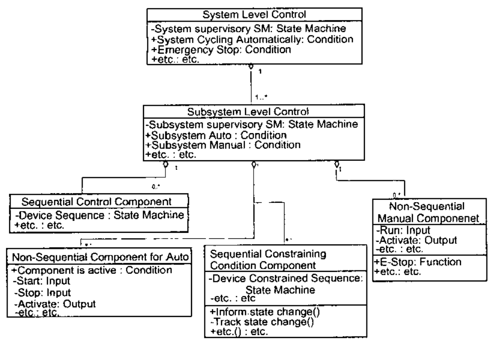
System Level Control
-System supervisory SM: State Machine
+System Cycling Automatically: Condition
+Emergency Stop: Condition
+etc.: etc.
1
1..*
Subsystem Level Control
-Subsystem supervisory SM: State Machine
+Subsystem Auto : Condition
+Subsystem Manual : Condition
+etc. : etc.
1
0..*
Sequential Control Component
-Device Sequence : State Machine
+etc. : etc.
Non-Sequential Component for Auto
+Component is active : Condition
-Start: Input
-Stop: Input
-Activate: Output
-etc.: etc.
Sequential Constraining Condition Component
-Device Constrained Sequence: State Machine
-etc. : etc
+Inform state change()
-Track state change()
+etc.() : etc.
Non-Sequential Manual Componenet
-Run: Input
-Activate: Output
-etc. : etc.
+E-Stop: Function
+etc.: etc.

Figure 5: Classes for work-cell control system

Based on the operations of various devices in the work-cell, four classes are necessary for such a system. The first class is *sequential control*, which defines a sequence of steps when the system is operated automatically. A component of this class contains nested device control state machines for both automatic mode and manual mode, which define the sequence in a subsystem-level auto state and manual state, respectively. When the subsystem is in the automatic mode, the automatic sequence state machine is selected. When the subsystem is in the manual mode, the manual sequence state machine is selected. The FSMs for components have only a partial observation of the system external to the controlled device. Typically, only interface states or conditions of components that the device interacts with, such as the destacker wishing to unload a part onto the press entry conveyor, are observed. Sequential control components can also contain supervisory state machines to single step or single cycle through the sequence.

The second class is *non-sequential auto*, which does not exhibit a sequence of steps while the system is running automatically. However, these components must be active to allow the subsystem to be in a condition where automatic behavior is possible. Typically, these are devices that supply energy such as hydraulic pumps, flywheels, motors connected to devices with clutches, etc. These must be running before the subsystem can be placed into an automatic operating mode.

The third class is *non-sequential manual*, which is very similar to the second. It typically controls the same devices, but in a mode which will not occur while in automatic operation such as running the devices in a reverse direction. Physically, this may be the same device, but from the model's point of view, it is treated as a different class due to its different characteristics. This limits the amount of complexity that would result in modeling both as a single class.

The fourth class is *sequential constraining condition*, which is primarily designed for tracking constraining conditions that can not always be determined directly from system sensors. A common constraint is allowing only one part at a time in a machine. For example, the press entry conveyor may not have more than one part in at a time, but there is no sensor to indicate this. The first three classes typically do not remember the state they were in when the controller is shut down while this class must. The requirement is forced by the fact that the constraining condition can not be directly determined from sensors. A FSM is used to keep track of the constraining condition and the states that indicate this condition are persistent. The first three classes will directly control the devices through outputs that they are associated with, while this class typically does not drive any outputs. The first three classes exhibit distinctly different automatic and manual behavior while this class does not. This class is primarily used as an interface coordinating components which are directly connected such as the destacker and the press.

The abstract class model that resulted from our analysis is fairly simple. Such simplicity is key to successfully implementing an application generator and its ability to be used by a domain expert (mechanical engineer) who is not familiar with control code generation. A simple model also enables the automatic generation of monitoring code since relatively few rules are needed for the limited finite set of different situations.

4 Control Software Construction and Experience

In this section, we present design of control code for the stamping/roll-forming work-cell using the abstract class model described in previous section and its implementation in PLC code. Figure 6 shows the implemented hierarchical object model based on the abstract class model.

At the system-level, a *wrapper line* object of the *system-level control* class is implemented to coordinate the entire system allowing automatic cycling, emergency stopping, emptying out of parts, etc. Three subsystem-level control objects, which correspond to the partitioning of the system shown in Figure 4, are implemented to control subsystems. Each of the three objects contains a supervisory FSM to allow manual or automatic behavior in each subsystem, as shown in Figure 7.

Each subsystem-level object controls one or more component-level objects. The destacker subsystem contains only the *sequence destacker* object of the *sequential control* class. Two nested state machines model this object's behavior. A manual sequence state machine, shown in Figure 8, is nested in the manual state of the destacker subsystem-level FSM to specify the manual operations. An automatic sequence state machine, shown in Figure 9, is nested in the automatic state of the destacker subsystem-level FSM to specify the automatic operations.

The press-transfers subsystem contains five component-level objects. The *sequence press/conveyors* object of the *sequential control component* class specifies the sequencing of the press, its entry and exit of conveyors. An automatic and a manual state machine controls the behavior for auto and manual mode, respectively. Similarly, the *sequence scissors lift* object of the *sequential control component* class specifies the sequencing of the scissors lift with its automatic and manual state machines. The hydraulic press must have its hydraulic motor running before it can be run automatically. This behavior is provided by the *run hydraulic motor* object of the *non-sequential component needed for auto* class. Similarly, the conveyors must be enabled before the subsystem can enter the automatic state. The *enable conveyor drives* object of the *non-sequential component needed for auto* class serves this purpose. Since only one part is allowed in the press subsystem at a time, and the part may be present but can not be detected by the sensors, a *track part position in press* object of the *sequential constraining component* class is used. A state machine is implemented in this object to keep track of the loading of a part in the press and its delivery out of the press subsystem.

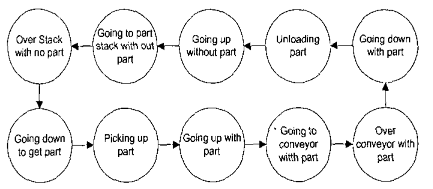

Figure 9: Automatic mode FSM

The roll former subsystem does not have a sequence (beyond simply running or being stopped), thus it does not contain any objects of the sequential control component class. This subsystem needs to control the roll former and its entry conveyor running in the forward direction to allow automatic behavior, therefore a *run roll former forward* object of the *non-sequential component needed for auto* class is used. To allow the roll former to be run in reverse when the subsystem is in manual, a *run roll former reverse* object of the *non-sequential manual component* class is used. As in the press subsystem, the presence of a part can not be determined by sensors, therefore a *track part position in roll former* object of the *sequential constraining condition component* class is needed.

The PLC code was initially manually coded based on the object model. We developed a prototype application generator based on the model to capture code design patterns from control engineers. This now allows a non-controls expert to interact with the application generator to specify the behavior of a different work-cell and have code automatically generated matching that which would have been written by a controls expert.

Based on our experience of constructing the appli-

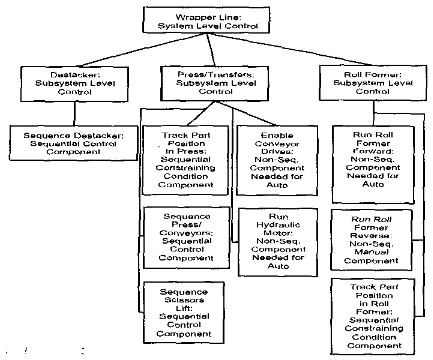
Wrapper Line: System Level Control
Destacker: Subsystem Level Control
Press/Transfers: Subsystem Level Control
Roll Former: Subsystem Level Control
Sequence Destacker: Sequential Control Component
Track Part Position in Press: Sequential Constraining Condition Component
Enable Conveyor Drives: Non-Seq. Component Needed for Auto
Run Roll Former Forward: Non-Seq. Component Needed for Auto
Sequence Press/ Conveyors: Sequential Control Component
Run Hydraulic Motor: Non-Seq. Component Needed for Auto
Run Roll Former Reverse: Non-Seq. Manual Component
Sequence Scissors Lift: Sequential Control Component
Track Part Position in Roll Former: Sequential Constraining Condition Component

Figure 6: Object model of the work-cell

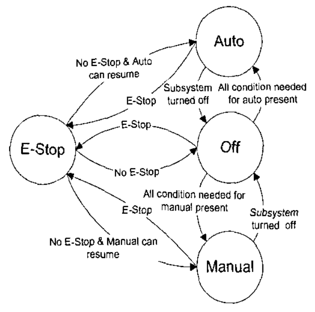
Auto
E-Stop
Off
Manual
No E-Stop & Auto can resume
Subsystem turned off
All condition needed for auto present
E-Stop
E-Stop
No E-Stop
All condition needed for manual present
E-Stop
Subsystem turned off
No E-Stop & Manual can resume

Figure 7: High-level FSM.

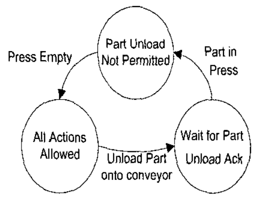

Figure 8: Manual mode FSM

cation, software development using the NSFM model shows several distinct advantages over the method used in current practices.

First, the development cycle is shortened and code functionality and quality are improved. The initial system design and code generation required approximately one half of the time. The typical process consists of the control designer obtaining an incomplete comprehension of the system requirement and starting to generate code. During the code generation, questions arise which result in a consultation with the customer (person specifying the function of the system). Frequently, new I/O is required and existing code has to be changed. Additional code is generated and new questions arise resulting in further consultation. Unfortunately, the customer usually does not comprehend the full system requirement at the beginning and the application code has to be created through a series of meetings. It is hard to remember all of the implications of the decisions that have been made along the way. The code ends up being a series of patches without the removal of obsolete and redundant pieces. The NSFM method forces a control engineer to obtain a full understanding of the requirements before any coding can begin since the NSFM must be fully specified. The NSFM provides a compact and complete model of the system behavior facilitating communication with the customer and removing ambiguity. Once the NSFMs are drawn, coding is a simple mechanical process. It can be written cleanly in one pass. The resultant quality and functionality is superior to the current practices in industry. Neither noticeable code size, nor execution speed penalties were observed in our application system.

Secondly, our approach reduces maintenance costs. The software debugging normally requires one third of the total development cycle. The programs based on the NSFM model provide an easily understood documentation of the system behavior and a standard pattern of code implementation simply correlates with it. For the same reasons, modification of the destacker sequence by someone other than the software developer also required one third of the normal time. An additional advantage was noted during the diagnosis of system failure, which was not comprehended by the implemented diagnostics. The hierarchical nesting of finite state machines provided a natural sequence of diagnostic steps. A technician can start checking the state machine from the top down by examining the current state, expected next state and conditions needed to achieve it. The location of an error where the missing condition exists can be quickly determined, and mapped to the physical condition. The transition from one state to the next typically requires examining only a few lines of code. The state machine method effectively eliminates the need to examine and understand large sections of code by automatically indicating the current state. Using a standard pattern for implementing the code allows a technician to quickly understand the functions and logic, and correlate it with the sequencing of the physical machine. Normally this is far more difficult in traditional control logic where the current state of the system can not be readily determined. There are no distinct state indicators. The state needs to be deduced by studying the conditions of the machine, the current value of the logical equations represented by the control logic and correlating the two. The logical equations represented by the code have usually undergone simplification making it more difficult to determine their functions.

Thirdly, the standard code pattern makes integration of diagnostic code far simpler. The reduction in effort is approximately one third. In addition, having the system context readily available from the FSMs provides data of better quality.

5 Conclusion

Our proposed methodology of NFSMs coupled with standard components was successfully applied in the domain of supervisory discrete event control of manufacturing work-cells. Initial development required one half the time; debug and later modifications required one third of the time.

A general abstract model and domain specific architecture was derived from implementing a stamping/roll forming work-cell. The abstract object model that resulted from our analysis is fairly simple. Such simplicity enabled us to successfully implement a prototype application generator, which allows non-controls personnel familiar with the function of a machine, to describe it and automatically generate PLC Ladder Logic control code. The application generator provides a model and architecture, which makes the automatic generation of context-aware monitoring code possible. Our intention is to use the application generator as part of a replicable experimental process for transferring software technologies from the theoretical domain into industrial applications.

The methodology has advantages even for manual code generation. Less time is required to initially generate the code with fewer errors and to debug the application software. It is easier to comprehend by someone other than the initial software developer and simpler to modify. It is also easier to diagnose a machine mechanical condition requiring corrective action.

References

[1] M. C. Paulk, B. Curtis, M. B. Chrissis, and C. V. Weber, "Capability of maturity model for software," Technical report, Software Engineering Institute, CMU, February 1993.

[2] D. K. Hammer, A. A. Hanish, and T. S. Dillon, "Modeling behavior and dependability of object-oriented real-time systems," *Computer Systems Science & Engineering*, vol. 13, no. 3, pp. 139–150, May 1998.

[3] B. Furht and W. A. Halang, "A survey of real-time computing systems," *International Journal of Mini and Microcomputers*, vol. 16, no. 3, pp. 141–155, 1994.

[4] S. Ren, G. Agah, and M. Saito, "A modular approach for programming distributed real-time systems," in *Lectures on Embedded Systems: European Educational Forum School on Embedded Systems (LNCS 1494)*, pp. 171–207, Springer-Verlag, Veldhovan, Netherland, November 1996.

[5] S. Birla and K. G. Shin, "Reconfiguration requirements for software to control automotive manufacturing machine tools," in *Proceedings of the 1998 Japan-USA Symposium on Flexible Automation*, July 1998.

[6] B. Kitchenham, P. Brereton, D. Budgen, S. Linkman, V. L. Almstrum, and S. L. Pfleeger, "Evaluation and assessment in software engineering," *Information and Software Technology*, vol. 39, no. 11, pp. 731–734, Novermber 1997.

[7] W. F. Tichy, "Should computer scientists experiment more?," *Computer*, vol. 31, no. 5, pp. 32–40, May 1998.

[8] S. Linkman and H. D. Rombach, "Experimentation as a vehicle for software technology trasfer — a family of software reading techniques," *Information and Software Technology*, vol. 39, no. 11, pp. 777–780, Novermber 1997.

[9] S. Wang, C. V. Ravishankar, and K. G. Shin, "Open architecture controller software for integration of machine tool monitoring," in *Proceedings of 1999 IEEE International Conference on Robotics and Automation (ICRA'99)*, pp. 1812–1820, Detroit, MI, May 1999.

[10] C. Shiu, M. J. Washburn, S. Wang, C. V. Ravishankar, and K. G. Shin, "Specifying reconfigurable control flow for open architecture controllers," in *Proceedings 1998 Japan-USA Symposium on Flexible Automation*, volume 2, pp. 659–666, Otsu, Japan, July 1998.

[11] OMAC Working Group. *OMAC API Documentation v0.23*, April 1999.