Constructing Reconfigurable Software for Machine Control Systems

Shige Wang, *Student Member, IEEE,* and Kang G. Shin, *Fellow, IEEE*

***Abstract*—Reconfigurable software is highly desired for automated machine tool control systems for low-cost products and short time to market. In this paper, we propose a software architecture based on a combination of object-oriented models and executable formal specifications. In this architecture, the machine control software is viewed as an integration of a set of reusable software components, each modeled with a set of event-based external interfaces for functional definitions, a control logic driver for execution of behavioral specifications, and a set of service protocols for platform adaptation. The behaviors of the entire software can be viewed as an integration of behaviors of components and their integration, and can be specified in a *Control Plan* specification language, which is based on *Nested Finite State Machines*, independently of the component implementations. Separation of structural specification from behavioral specification enables the controller software structure to be reconfigured independently of application, and software behavior to be reconfigured independently of controller software structure. When the system needs reconfiguration due to changes in either application requirements or the execution platform, the software with our architecture can then be reconfigured by changing reusable components and their interactions in structure for functional capability, and by changing the Control Plan program for behavior. Both types of reconfiguration can be done at the executable code level after the software is implemented. The proposed architecture also supports reconfigurability to facilitate heterogeneous implementations and vendor-neutral products. Our evaluation based on current software construction practices for both laboratory machine tools and an industry machining system has shown that the goals of higher reconfigurability and lower development and maintenance costs are achieved with the control software constructed using the proposed architecture.**

***Index Terms*—Controller architecture, machine control systems, reconfigurable software, software models.**

I. Introduction

AS MORE ADVANCED and cheaper hardware becomes available and applications get complicated with shorter time-to-market demands, software for today's machine control systems must support agile system reconfiguration with different combinations of hardware and software [6], [15], [17]. Software for machine control systems is usually designed and implemented with a set of components, such as device drivers, control functions, and algorithms, all running on a designated platform. Components may need to be added, removed, and replaced to satisfy new product requirements during the software life cycle. The execution platform may also need to be upgraded, oftentimes with new computing and communication hardware and software. This trend calls for reconfigurable software that reuses existing software components to generate the control software for new hardware and applications very quickly. This will, in turn, enable low-cost product development at the manufacturing level and short time to market at the enterprise level.

However, the reconfigurability of software for current machine control systems is very limited, although the concept of component-based software integration [28] has already been adopted in controller software development. Specifically, the following limitations in current control software development practices hinder the reconfigurability of software.

1) *Application software is partitioned and implemented with proprietary information.* For example, a device driver for a machine tool without monitoring, and that with monitoring, are usually implemented with very different interfaces, thus making it difficult to reconfigure software for controller without monitoring to one with monitoring when such a function is required as the application evolves. This difficulty is due mainly to component implementations based on the traditional top-down system partitioning, which requires components and their interactions to be fully specified before their implementation. Components for reconfigurable software cannot be implemented in this way, as external interactions in different configurations are not always known when a component is implemented.
2) *Control behaviors of the software are either built inside the implementation and hence, not customizable, or not modularized and associated with the corresponding software components.* Examples of the first case are hard-coded system startup and shutdown procedures. Software for a programmable logic controller (PLC) requiring global shared information is an example for the second case. The root cause of this problem is that the selection and implementation of control logic are usually determined by the application requirements or physical processes, while the selection and implementation of components are determined by the physical machine setup. The architectural mechanisms in current machine control software do not support separation of development of component-level behaviors from that of component-level functions.

Manuscript received December 5, 2001. This paper was recommended for publication by Associate Editor G. Menga and Editor N. Viswanadham upon evaluation of the reviewers' comments. This work was supported in part by the Defense Advanced Research Project Agency (DARPA) under US Airforce Contract F33615-00-C-1706.

The authors are with the Real-Time Computing Laboratory, Department of Electrical Engineering and Computer Science, The University of Michigan, Ann Arbor, MI 48019-2122 USA (e-mail: wangsg@eecs.umich.edu; kgshin@eecs.umich.edu).

Digital Object Identifier 10.1109/TRA.2002.802235

3) *Software implementation is specific to platform configuration.* Currently, software development for machine control systems requires full knowledge of the execution platform configuration before the software is implemented. The software components are implemented with configuration information, such as the number of inputs–outputs and their locations, the number and type of processors for execution, communication channels, and protocols for information exchange. However, the original platform configuration is very likely to change many times during the system's lifetime or as the application requirements change, thus forcing the software components to be reimplemented. It is also difficult to construct software using existing components from different applications owing to differences in their requirements from execution environments.[1]

To address the aforementioned issues, we propose a software architecture especially for constructing reconfigurable software with reusable components. Our goal is to enable reuse of implemented software components across different applications and different platform configurations, and behavior reconfigurability at the executable code level. In this architecture, reusable software components are modeled with a set of event-based external interfaces, a set of communication ports, a control logic driver, and service protocols. Components can be structurally composed by linking their communication ports and then be mapped to a platform by customizing their service protocols. The control logic, called *behavior*, of each component is modeled as a *Nested Finite-State Machine* (NFSM), which is a formal model for behavior compositions. Such control logic in the NFSM can then be specified in table form and executed by the control logic driver inside each component. Both system- and component-level behaviors can be specified in a *Control Plan* program, which is an executable specification used for both design-time analysis and run-time execution. Since the behaviors and functions are separated in our architecture, nonfunctional constraints such as timing and resource constraints can be analyzed at an early phase of development, as system behaviors normally have more significant effects on these constraints than system functionalities and platform configurations.

Our main contribution in this paper lies in separating concerns in controller software development at the software architecture level, so that different aspects of controller software can be configured/reconfigured independently and after implementation. In our architecture, structural and behavioral reconfiguration as well as platform reconfiguration are separated from each other and can be done independently of each other. This enables experts in one area (e.g., control logic design) to work independently on one aspect of a controller without having to know the other areas (e.g., software modeling and hardware integration). The support for postimplementation reconfiguration reduces development costs as compared with those methods being used in current software engineering practice that focus on the preimplementation design phase and require reimplementation upon reconfiguration. The other contributions of this paper include construction and evaluation of reconfigurable software for a real control system and showing/making tradeoffs between flexibility and performance.

[1]These problems are known as "architecture mismatches" in software engineering [34].

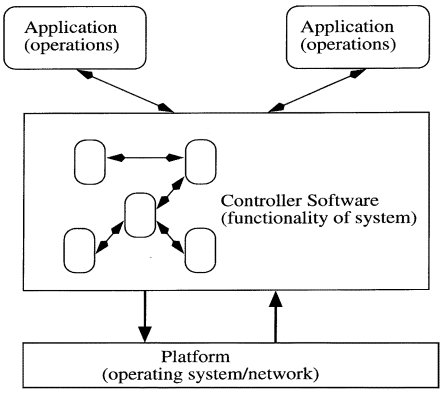


Fig. 1. Reconfigurable software structure.

The rest of this paper is organized as follows. Section II describes the architecture for construction of reconfigurable software, including component structure, composition model, and structural reconfiguration. Section III describes the NFSM model, specifications in *Control Plan*, and behavior reconfiguration. Section IV presents our evaluation results based on control software construction for two laboratory machine control systems and an industry manufacturing workcell. Section V summarizes the related work, and the paper concludes with Section VI.

II. Software Architecture

Reconfiguration of a machine control system can be abstractly viewed as changes to *application process* and/or *physical configuration*. The former determines the control algorithms, operations, and their sequences required to manufacture a product, while the latter defines the (machine tool and computation) devices and their functionalities. Thus, we divide the software for machine control systems into two parts: *controller software* and *application specifications*, as shown in Fig. 1. The controller software consists of reusable software components corresponding to the physical machine configuration and defines only the functionality of the machine control system. The application specifications describe the product process as well as the desired control logic of the controller software. In our architecture, the controller software is expressed as a composition of communicating software components, and the application specifications are expressed as the integrated behaviors in a Control Plan program. Such a software structure breaks the dependency between the application and the physical machine configuration and, therefore, higher reconfigurability can be achieved by supporting the same application specification executed on different controllers, and supporting the same controller software executing different application specifications. The controller software needs to be reconfigured only when the physical configuration changes, and the application specifications need to be altered when the product requirements change.

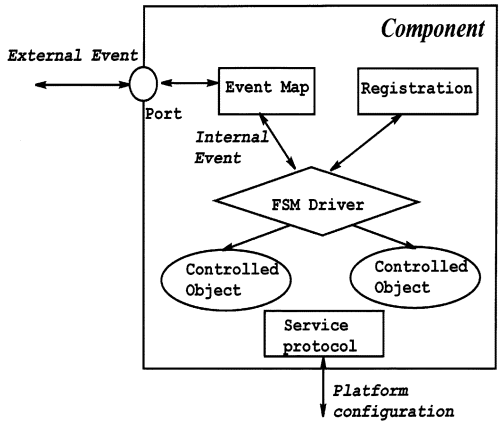


Fig. 2. Reusable component structure.

A. Component Structure

Components are preimplemented software modules (or already instantiated but customizable objects within an object-oriented model) and are used as building blocks to construct the controller software. A component defines the functionality of a device or subsystem, which can be as simple as an I/O device like a position sensor, or a control algorithm like proportional-integral-derivative (PID) control, or as complex as a composed subsystem like coordinated axes.

The structure of a software component includes a set of event-based external interfaces with registration and mapping mechanisms, communication ports, a control logic driver, and service protocols, as shown in Fig. 2.

Event-Based External Interfaces: External interfaces are designed to expose component functionalities to the external world, i.e., define operations that can be invoked from outside. External interfaces in our architecture are represented as a set of acceptable global (external) events with designated parameters. Event-based interfaces enable operations to be scheduled and ordered adaptively in distributed and parallel environments and allow components to be integrated, at executable code level, into the system. A customizable event-mapping mechanism is devised and added in each component to achieve the translation between global events and the component's internal representations. Such a mapping separates a component implementation from its interfaces, thus making multiple implementations of one operation possible. Since the mapping is internal, it can be customized without knowing interactions with other components. A registration mechanism is also added to perform run-time checking on the validity of received events. A user can manage which operations are allowed to be called by customizing the registered events. Only those operations invoked by authorized and acceptable events will be executed. These customizations can be stored as predefined files in a local system for run-time loading, or selected by a user when the system starts up.

In an integrated system, a set of components may interact with each other to obtain the desired services (e.g., data transformation and control-command generation). The services that each component provides are specified as acceptable external events, and other components can invoke the desired functions only through sending the corresponding events to the target components. Such relationships are constructed during design time and can be changed only at some predefined safe state (e.g., configuration state) during run time. This makes our component model different from commonly used models such as the common object request broker architecture (CORBA), the distributed component object model (DCOM), and Jini [27], which are usually based on remote procedure calls and heavily depend on predefined middleware services such as naming and lookup services. Therefore, the control software constructed with our component model consumes less computation resources and supports more predictable execution, so it can be implemented on inexpensive hardware while preserving reconfigurability that is difficult to achieve with a fixed implementation. On the other hand, middleware services can be integrated in the system as components, when necessary, by mapping the service invocations to a set of events corresponding services.

Communication Ports: Communication ports are used to connect components for integration. They are physical interfaces of a component, and are the only mechanism by which components interact with each other. Each communication port has a set of attributes associated with it which define the type of communication port (send only, receive only, or both, buffered or nonbuffered), message-exchange methods (shared, queued, or immediate), the way of communication (synchronous or asynchronous), message-delivery policies (first-in-first-out, priority-based), and conflict-resolution policies (overwritten AND, OR). Proper values of these attributes can be selected during design time through analyses.

A pair of communication ports form a communication channel. In this regard, the communication port is similar to the one defined in Real-Time Object-Oriented Model (ROOM) [23]. However, our communication port is not only an abstract model representation, but also it can be an implemented mechanism in a final product to support reconfiguration, i.e., it is *designed for reuse and reconfiguration*. Hence, a port is not implemented as some abstract class, but as a parameterized object, and different instantiations differ only in their parameters as described in [28]. Different protocol levels can be implemented by customizing the communication ports. For example, the application-level protocol can be implemented by defining which event list is acceptable for a port, while the underlying infrastructure-level protocol can be implemented by customizing the attributes of the communication port. Each reusable component can have one or more communication ports. The number of ports that a component needed in a configuration can be determined by the system integrator. Ports can be customized with different service protocols to meet different performance requirements. Multiple connections can also share one communication port.

Control Logic Driver: The control logic driver, also called the *finite-state machine (FSM) driver*, is designed to separate function definitions from control logic specifications and to support control logic reconfiguration at executable code level. The control logic driver can be viewed as an interface to access and modify the control logic inside a component, which is traditionally hard coded in the component implementation. Every

component that executes some control logic should have such a driver inside itself. The control logic of a component can then be fully specified as a state table [30] for execution. A control logic driver will generate commands to invoke operations of the controlled objects at run time according to its state table and received events. State tables can also be packed as data and passed to another component to reconfigure the receiver component's behavior remotely.

The control logic driver of a component is a center piece to enable postimplementation reconfiguration of component behavior. It invokes the control object functions based on current component state and incoming events to communication ports, which are specified by state table entries. Function calls are statically bound to internal events at implementation time, and the control logic driver invokes the control object functions by generating internal events for the corresponding control objects. For each component, multiple state tables can be designed to specify different desired behaviors in different system modes. However, only one state table for a component can be active at a time.

Although the control logic driver introduces additional overhead to the system as more steps a component has to go through to invoke an operation, such processing and related bindings are statically configured before normal execution. Therefore, the overhead introduced by the control logic driver should be negligible to the application-level performance; it may be significant at low level due to frequent long jumps and pipeline flushes caused by the sequence of function calls to process state transitions. On the other hand, efforts and costs for software reconfiguration when application requirements change, overweigh performance for resource-rich systems such as PC-based controllers.

Service Protocols: Service protocols define execution environments or infrastructures of a component. They are designed to make components adaptive to different platforms. In our architecture, we assume that the underlying infrastructures provide unified interfaces for different types of services as defined in the portable operating system interface (POSIX) [10] and, therefore, form a virtual machine for application-level components of a controller. The service protocols are used to customize such a virtual machine so that only those required services will be integrated into the system. For example, a component can specify its communication mechanism as a message queue or shared memory. Examples of service protocols include scheduling policies, interprocess communication mechanisms, and network protocols.

Service protocols are implemented as a set of attributes of a component. Selection of services is implemented by assigning the desired values for the service attributes. Such selection is based on the mechanisms available on a given platform and performance (such as timing and resource) constraints of a system. The selected services will be bound to the corresponding function calls provided by the infrastructures either statically or during the software initialization.

There is also another type of component commonly used in controller software, which only handles computation without any control logic. The functionality of such a component can be considered, for example, as transforming data from one format to another. We model such components, called *computational components*, with the same structure as above, but without a control logic driver. Since computational components are usually used under the control of another component with control logic, they can be treated as a black box with input and output data associated with some special events (e.g., IN_DATA and OUT_DATA) during component integration and analysis.

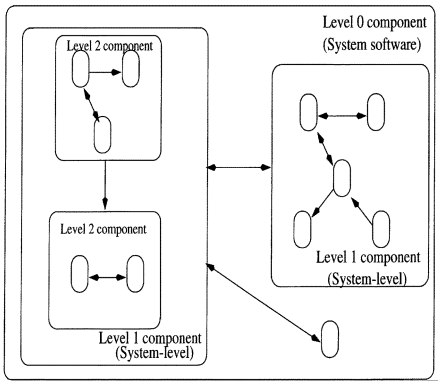


Fig. 3. Hierarchical composition model.

At run time, each component retrieves events from its communication ports either upon some event arrival or periodically. The acceptable events are then translated to internal events and fed to the control logic driver. The control logic driver determines desired actions and output events and sends local commands to corresponding controlled objects. After receiving commands, the controlled objects in a component will perform some actions (one or a sequence of function calls) and generate results. However, the controlled objects are not allowed to feed events back to control logic driver to prevent local cycles.

B. Composition Model and Structural Reconfiguration

Structurally, the controller software can be constructed by integrating a set of preimplemented components, each with the structure shown in Fig. 2. Interactions among the constituent components can be defined as events to be exchanged via designated communication ports. To facilitate software construction, a set of components can be preferred to be composed as one large complex component during high-level integration. Our component model supports composability. Components can be organized hierarchically in our composition model to support reconfiguration with different component granularities. A high-level component may consist of a set of communicating low-level components, as shown in Fig. 3, with a high-level control logic driver, communication ports, and customizable service protocols. Such composition implies the hierarchical behavior where the behaviors of inner (lower level) components are part of the behavior of the component that contains them.

While such a composition provides reusability and reconfigurability at multiple granularity levels, the overheads may increase as the component hierarchy becomes deeper. Since the functionality of a subsystem in a control system is relatively static, and reconfiguration only happens among a certain range of levels, the overhead can be lowered by limiting the number of granularity levels during design and integration.

The controller software constructed with such reusable components is highly reconfigurable. Reconfiguration of controller software involves only structural changes, including component additions and removals, component replacements, and system reorganizations. Component additions and removals are necessary when new devices, control functions, and control algorithms are introduced into the system. Such reconfiguration can be done by adding or removing the corresponding components and linking communication ports at a certain granularity level. For component additions, the modification of an existing system is minimized if the new added components can use the communication ports of other existing components. To enable their interaction with other existing components, the event mapping and registration mechanisms of new added components may need to be customized. On the other hand, removal of a component requires updating the set of acceptable events in components that are used to communicate with the removed component.

When an existing device, control algorithm, or hardware/software in the platform is replaced, the corresponding components may need to be replaced. Such replacement can be viewed as a component removal followed by a component addition. If the added component has the same configuration as the removed one (for example, the number and type of communication ports), they can be easily switched in and out. Otherwise, the new component and the components it communicates with need to be reconfigured. Event mapping and registration information may also need to be changed when a different component implementation is used.

Another type of common reconfiguration is called *system reorganization*, which requires changes of components' interactions or their execution environments, but not the components themselves. System reorganization usually occurs when the relationships among subsystems change (e.g., forming a system with two coordinated axes and one independent axis as a system with three coordinate axes), or when platform configuration changes (e.g., allocating an axis component to a separate processor communicating with the other two axes components on another processor through Ethernet instead of the original three axes executing on a single processor communicating through shared variables). With our component structure and composition model, reorganization of component relationships can be achieved by modifying the corresponding communication port linkages, and reorganization of platform configuration can be achieved by customizing the service protocols of the involved components.

Although dynamic structure reconfiguration of software is widely used in many general-purpose applications and supported by many platforms, such as Windows dynamic link libraries, DCOM interface query, and Jini lookup service, we limit the online structure reconfiguration to be only parameter adjustments during normal execution, to minimize the unpredictability introduced by these dynamic reconfiguration mechanisms. Structure reconfiguration other than parameter adjustment, including addition, removal, replacement of software components (e.g., device drivers and control algorithms), and system reorganization, has to be done statically either before the software is loaded for execution or in some special states at which the system is not subject to timing constraints, e.g., configuration or toolchange state.

III. Behavior Specification and Reconfiguration

The correctness of controller software not only depends on the structure of the software (components and their interactions), but also on the behaviors of the software. While the structure composition of components defines the functional capability of controller software, it is the behavior of the software that defines the dynamic properties of the system. In our architecture, the overall software behaviors can be viewed as an integration of behaviors of components. With the control logic driver in each component, the behavior of a component can be specified and verified separately before its integration into the final system. Since control applications are normally time critical and safety critical, and require software behaviors to be analyzed thoroughly before implementation, specification and verification methods based on formal methods are highly desired. Modularized behaviors are also required to accomplish behavior reconfiguration when the system is reconfigured structurally. To this end, we used NFSMs to model and verify the behaviors of components and their integration, and developed the Control Plan constructs for behavior specifications. Our specification can be directly executed by the control logic drivers. Therefore, specification-based early-phase simulation and evaluation become possible, and the errors introduced by implementing the specification in some programming language can be minimized. Moreover, such specifications are formally verified. The model and specification of behavior also extend the specification methods that are currently being used in industry, such as IEC-1131, by modularizing and supporting fine-granularity specifications at component level. Different implementations of the controller software can then be selected to execute the same behavior specified in Control Plan programs to satisfy nonfunctional constraints. Consequently, behavior reconfiguration can be achieved separately from the control software implementation by changing the Control Plan program.[2]

A. Behavior Specification

The behavior specifications of control software are divided into two disjoint parts: control logic specifications and operation sequence specifications.

Control Logic Specifications: Define the static part of software behavior or the control logic of a component. It is modeled as an NFSM with a set of traditional "flat" FSMs organized hierarchically. We use the Mealy machine [30] for each FSM in a NFSM. A NFSM at level i, M_i, can be defined as

$$M_i = \langle S_i, I_i, O_i, T_i, s_{i_0} \rangle \text{ (level-}i \text{ FSM)}$$

where S_i is a set of states of the ith level FSM, I_i and O_i are a set of inputs and outputs, respectively, T_i is a set of transitions,

[2]The behavior reconfiguration associated with computational components is not considered here, due to the fact that changes of computation behaviors imply using a different computation equation or algorithm, which is normally implemented as a different component. Reconfiguration can, therefore, be achieved by structural reconfiguration.

and s_{i_0} is the initial state of M_i. A noninitial state of M_i may contain a set of FSMs at the $i+1$th level.

The NFSM behavior model corresponds to the hierarchical composition model as described in Section II-B. Only FSMs of top-level components in a composition are visible during behavior configuration. A control logic change in a component only affects the FSMs that immediately connect to it in a composition.

The FSM of a component can be fully specified in a table with each entry defining a possible transition. The structure of each entry is

$$\text{STATE}, \text{EVENT}_{\text{input}}, \text{ACTION\_LIST}, \text{ STATE}_{\text{next}}$$

where STATE is the current state of the system, $\text{EVENT}_{\text{input}}$ is an input event, ACTION_LIST specifies the actions to take or the functions to call, and $\text{STATE}_{\text{next}}$ is the component state after the transition. STATE and $\text{EVENT}_{\text{input}}$ together determine an entry in a state table uniquely, thus determining a unique set of operations and a unique next state. For simplicity, we assume there exists a state variable in each component. STATE in a state table only enumerates the possible values of the variable.[3]

Operation Specifications: Define the desired run-time input sequence that will trigger a designated sequence of operations if there are no other conflicting commands from higher-priority sources such as a human operator or an agent program for emergency cases. An operation sequence is specified as a preprogrammed event sequence consisting of a list of rows, each of which is in the format of

$$[\text{WHEN } state]\,[\text{INPUT } e_{\text{input}}\,[\text{PARAM } parameter]]\quad \text{OUTPUT } e_{\text{output}}\,[\text{PARAM } parameter]$$

where *state* is the current state, e_{input} is the received event, e_{output} is the event to send out, and *parameter* is the data attached to the corresponding event and is treated as a data chunk in the specification.

Although events used in an operation specification are normally global events for reasons of portability and reusability, internal events of a component can be used in the component's operation specification when the operation specification is attached as a parameter to some global event for the component.

Specifications in Control Plan: A Control Plan specifies software behaviors and consists of *logic definitions* and *operation specifications*, corresponding to the control logic specifications and operation sequence specifications, respectively. The structure of a control plan is shown in Fig. 4.

A FSM-ENDFSM block specifies a state table while an OPERATION-ENDOPERATION block specifies a designed operation sequence for a component indicated by label. The location is an option that indicates where the block will be executed. A block will be executed at the current local site by default if the location is not specified.

It is possible for a Control Plan to have multiple FSM and OPERATION blocks for a component for run-time reconfiguration. A block can also be attached to an event as data to pass around. Details can be found in [33].

[3]It is easy to implement the state variable as some combination of a set of local variables, while values of the state variable are logical combinations of values of local variables. Events can be processed similarly.

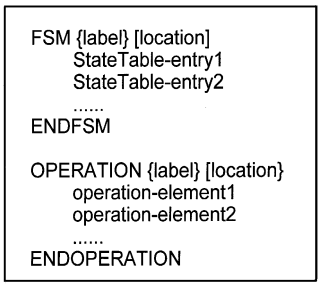


Fig. 4. Structure of a Control Plan program.

Specifications in Other Models: In a complex integrated machine control system, different subsystems may deal with different processes and, therefore, require different models for their behaviors. Behavior specifications in other models or languages can be converted to a Control Plan using translators. Translators are the programs designed to convert different models and specifications in a system to Control Plan programs. They are domain specific and specification language-dependent, meaning that each translator can only convert programs in a designated specification language to Control Plan programs. Therefore, several translators may be required in a system if there are multiple programs written in different specification languages.

The behavior specifications of controller software usually have to be verified before execution for safety purposes. Some tools and methods [1], [7] have been developed to check the properties of an NFSM, such as liveness and deadlock. Since we focus only on the software architecture, we assume the behavior specifications given by the designer for execution are correct.

B. Behavioral Reconfiguration

Behavioral reconfigurations include changes in control logic and operation sequence. A control logic reconfiguration is required when a component needs to process inputs differently. Such reconfiguration can be achieved by defining a different state table. The control logic driver in each component enables the same component to execute different behaviors by loading different state tables and operation-sequence specifications. An operation sequence needs to be modified when the machine operation procedure changes (e.g., use the same machine to manufacture parts of another product). Such reconfiguration can be achieved by defining a new operation sequence. Then, a Control Plan program with new specified state tables and operation sequences can be generated, stored in a local subsystem or elsewhere remotely, and loaded into the system during the configuration/reconfiguration phase at run time. Thus, new behaviors can be achieved at executable code level without regenerating configurations of controller software and implementations of components.

The behavior specifications can be classified further as device dependent and device-independent behaviors. The device-independent behaviors depend only on the application-level control logic and can be reused for the same application with different

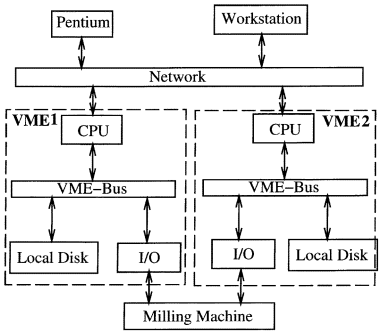


Fig. 5. Evaluation testbed configuration.

devices. The device-dependent behaviors are specific for a device or a configuration, and can be reused for different applications with the same device.

Both structure composition and behavior integration of components are required in a machine tool controller implementation. The integration of structure and behavior is done through loading a Control Plan to the corresponding components, either statically at integration phase or dynamically at run time. During execution, the control logic driver of each active component will invoke the local operations according to the FSM specified in Control Plan and the incoming events at their communication ports. Although the structural configuration and reconfiguration have to be done statically, the behavioral reconfiguration can be done online upon request.

IV. Evaluation

We evaluated the proposed architecture by reconfiguring an existing machine tool control software for different applications on our laboratory testbed, as shown in Fig. 5. The software executes on two control computers (with their own processors and memory) running the QNX real-time operating system. Such configuration is based on analysis of system workload and timing requirements of the control system. There is also a PC with Pentium processor running Windows NT for human graphic interfaces and a SUN Workstation running SunOS for offline control logic development and data analysis. These devices are connected through peer-to-peer Ethernet. The evaluation metrics include the number of component and behavior modifications needed to meet new requirements, the amount of effort required to accomplish the reconfiguration, and the run-time overheads introduced by the new architecture. Fewer modifications and less effort indicate better reconfigurability, while the run-time overheads indicate the efficiency of the software. Besides the laboratory prototypes, we also evaluated the architecture in an industry setting by constructing a real machine control system.

A. Software Reconfiguration for Machine Tool Controller

We have developed motion-control software for the controller of a three-axis milling machine [35], which dynamically coordinates the motion of three axes. We first reimplemented the software using the proposed architecture. The new software structure includes control algorithms, physical device drivers, and a coordination subsystem, as shown in Fig. 6. Some high-level components used in the existing controller are as follows.

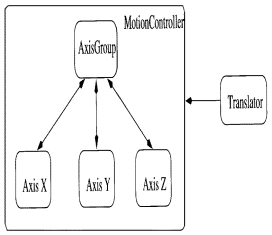


Fig. 6. Structure of the Robotool motion-control software.

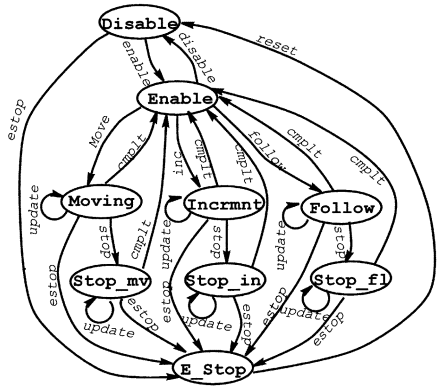


Fig. 7. Axis FSM.

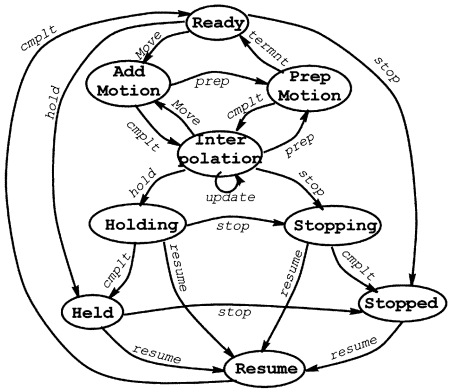


Fig. 8. AxisGroup FSM.

1) *AxisGroup:* receives a process model from the user or predefined control programs and coordinates the motion of the three axes by sending them the corresponding setpoints.
2) *Axis:* receives setpoints from AxisGroup and sends out the drive signal to the physical device according to the selected control algorithm (PID or FUZZY).
3) *G-code Translator:* translates a G-code program into a Control Plan.

The test application is a sequence of milling operations. The system behaviors are specified as an overall machine-level FSM, FSMs for Axis and AxisGroup components, and a G-code program for operation sequences, shown in Figs. 7–10.

Reconfiguration With Force Supervisory Control: Our first reconfiguration is adding a force supervisory control algorithm into the controller. This algorithm was developed by engineers

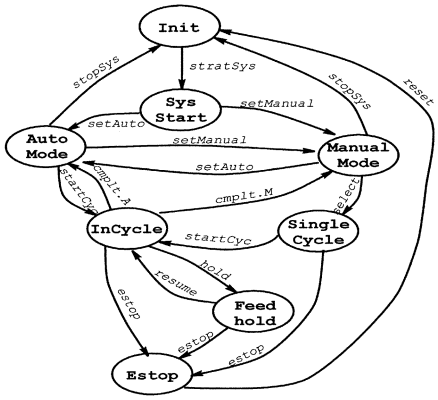


Fig. 9. Machine-level FSM.

```
G-code program:
n10 g01 x10 y10 z0 f1
n20 g01 x10 y10 z5 f0.5
n30 g01 x30 y10 z5 f1
n40 g01 x0  y0  z0 f5

Translated CP:
WHEN AutoMode OUTPUT startCyc PARAM (0,0,0,10,10,0,1)
WHEN InCycle  INPUT  cmplt    OUTPUT startCyc PARAM (10,10,0,10,10,5,0.5)
WHEN InCycle  INPUT  cmplt    OUTPUT startCyc PARAM (10,10,5,30,10,5,1)
WHEN InCycle  INPUT  cmplt    OUTPUT startCyc PARAM (10,10,5,0,0,0,5)
```

Fig. 10. G-code and control plan.

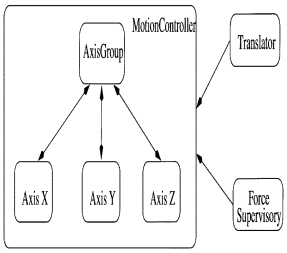


Fig. 11. Software after reconfiguration with supervisory force control.

at The University of Michigan, Ann Arbor, to compute a new feedrate to override the initial assigned feedrate based on the forces sensed at run time, and is implemented as a computational component. The reconfiguration was achieved by customizing the motion-controller software with an additional port to communicate with the force supervisory control component, as shown in Fig. 11. Since the force supervisory control algorithm only does the computation, no behavior change is required during this reconfiguration.

Reconfiguration With Broken-Tool Detection: The broken-tool-detection algorithm is another one developed by mechanical engineers to detect abnormal forces at run time, and send a stop signal to the motion controller when such a force is observed. A broken-tool-detection algorithm was developed separately and implemented as another component. The structural reconfiguration is achieved by linking the communication port of the broken-tool-detection component to the motion controller

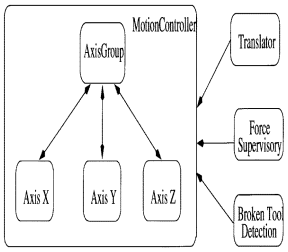


Fig. 12. Software after reconfiguration with broken-tool detection.

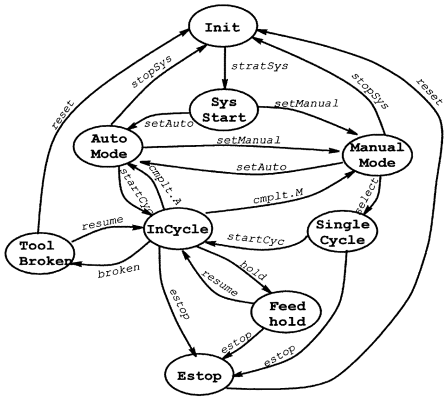


Fig. 13. Machine-level FSM with broken-tool detection.

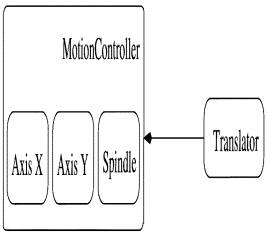


Fig. 14. Software structure of the RMT motion controller.

communication port for commands.[4] The software structure after this reconfiguration is shown in Fig. 12.

Alternatively, the broken-tool-detection component introduces behavioral reconfiguration of control logic at the machine level. This is achieved by changing the machine-level FSM, as shown in Fig. 13, while keeping the rest of behavioral specifications intact.

Reconfiguration for RMT Machine: We also developed software to control the motion of Reconfigurable Machine Tool (RMT), which is a modularized and composable two-axis machine built by engineers at The University of Michigan. The current RMT has neither coordinated motion nor monitoring. Due to the similarity of physical devices and behaviors of RMT and Robotool, we constructed software for the RMT controller by reconfiguring the Robotool control software. This is achieved by removing coordination- and monitoring-related components and adding a new *Spindle* component to control a discrete device on RMT that has only two positions. Fig. 14 shows the software structure for the RMT controller.

The behavior specifications for Axis components are the same as those for Robotool. However, the machine-level

[4]In our case, the broken-tool-detection component shares the same communication port with other agents that send commands to the motion controller. The control software does not tell who sends each command.

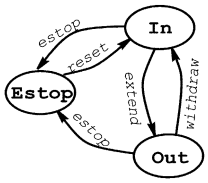


Fig. 15. FSM for the spindle component.

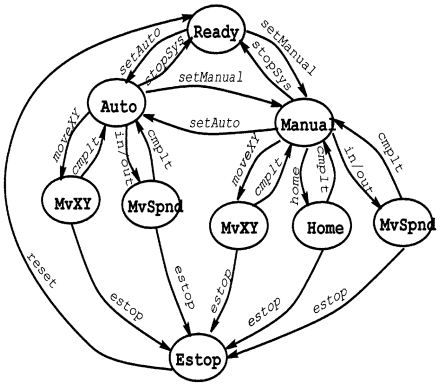


Fig. 16. The machine-level FSM for the RMT.

```
G-code program:                 Translated CP:

n10 g01 x10 y10 f1              WHEN Auto  OUTPUT moveXY PARAM (0,0,10,10,1)
n20 g21                         WHEN MvXY INPUT cmplt OUTPUT extend
n30 g22                         WHEN MvSpnd INPUT cmplt OUTPUT withdraw
n40 g01 x20 y10 f1              WHEN MvSpnd INPUT cmplt OUTPUT moveXY PARAM (10,10,20,10,1)
n50 g21                         WHEN MvXY INPUT cmplt OUTPUT extend
n60 g21                         WHEN MvSpnd INPUT cmplt OUTPUT withdraw
n70 g01 x0 y0 f5                WHEN MvSpnd INPUT cmplt OUTPUT moveXY PARAM (20,10,0,0,5)
```

Fig. 17. G-code program and control plan for RMT.

behavior needs to be reconfigured due to the new lower-level behavior changes introduced by the Spindle component. A different G-code program is used for a new operation sequence for manufacturing four- or eight-cylinder engine blocks. All these behavioral reconfigurations are accomplished by creating new state tables and Control Plan programs, which are developed and verified separately from the controller software construction by control engineers. Figs. 15–17 show these new behavioral specifications.

B. Evaluation Results

For all the reconfiguration cases examined, the machines operated correctly and manufactured desired products after the necessary reconfigurations.

Our experiences in constructing these controllers' software have shown that the proposed architecture provides significantly improved reusability and reconfigurability of integrated software. The component structure separates the functionality definition from the behavior specification, so that a component can be reused for different applications. Furthermore, the behavior is separated from the component implementation and specified in a NFSM-based Control Plan so that the system developer who may not be familiar with the software can also do behavioral reconfiguration and analysis on a daily basis. Separation of device-independent behaviors from device-dependent ones further enhanced the reusability of behaviors. As seen in the above cases, the Axis behaviors can be reused when the component is reused in a different application.

TABLE I
MODIFICATIONS NEEDED FOR EACH RECONFIGURATION

	# components	# reconfigured	# behavior	# reconfigured
Original Robotool SW	52	-	6	-
SW w/ force supvr	53	3	6	0
SW w/ broken tool	54	2	6	1
SW for RMT	37	3	5	2

TABLE II
EFFORTS NEEDED FOR RECONFIGURATIONS (IN HUMAN- MONTH)

	Traditional approach	Proposed approach
SW w/ force supvr	2.2	0.6
SW w/ broken tool	2.0	0.8
SW for RMT	8	4

Table I illustrates the numbers of components and behavioral specifications modified for each reconfiguration.[5] Only 3 among 37 (< 9%) components of Robotool software need to be modified to construct control software for RMT, while less than 3 among 54 (< 6%) components need to be changed for the reconfiguration with force supervisory control and broken-tool detection. On the other hand, the behavioral specifications required higher percentage modifications because a low-level behavioral change can trigger a cascade of changes at higher levels. However, since the behavior specifications and component functional specifications have been separated, the behavior changes did not affect the integration of components and can be done easily without requiring much of programming skills. The numbers in the table are also consistent with the fact that functions are more stable than the behaviors for applications in the same domain. The small number of components and behaviors that need to be modified indicates reduction of software complexity.

As illustrated in Table II, it also took less effort to reconfigure existing software for a new application. As compared to the traditional *ad hoc* implementations, reconfiguration took only about one third of the effort for integrating force supervisory control and broken-tool detection, and the time spent on the RMT software construction was also reduced by 50%.

Although the proposed architecture demonstrated high reconfigurability, we observed some performance penalties associated with it. Table III shows the computation time for the

[5]The components referred to in the table are those at levels above the Axis in the composition hierarchy.

TABLE III
COMPUTATION TIME FOR SOFTWARE WITH DIFFERENT APPROACHES (IN MILLISECONDS)

	Traditional approach	Proposed approach
Robotool SW RMT SW	2.1 1.3	3.0 1.5

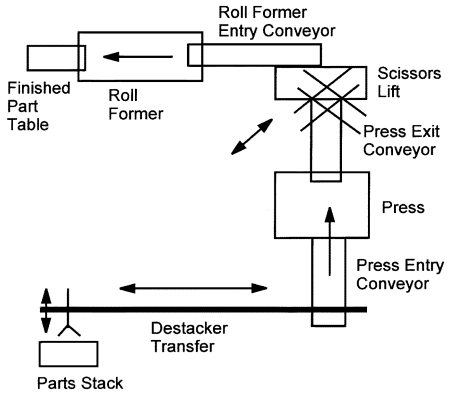


Fig. 18. Workcell configuration.

MotionController component,[6] collected by a custom-designed hardware component, *VMEStopWatch card*, which has a built-in high-resolution timer (25 ns). The extra overhead in our approach can come from the event-processing mechanisms inside the component, including authority checking, external event to internal event mapping and control logic driver overhead (FSM state table lookup and controlled object function invocation) and component communication mechanism. A further breakdown analysis showed that the overhead introduced by communication dominates the overall overhead. This suggests that reducing levels of hierarchy and increasing component granularity will yield more effecient code. The issues related to the tradeoffs between reconfigurability and run-time efficiency need further investigation.

C. Results of Industry Applications

To further evaluate the proposed architecture for real industry applications, we collaborated with our local industry partner to implement a stamping/roll-forming workcell, as shown in Fig. 18. The workcell was partitioned into three subsystems with a set of FSMs, some of which are shown in Figs. 19–21. The preliminary evaluation results showed that the proposed architecture had shortened the software development time by the factor of two, reduced the software debugging time, and made the integration of diagnostic code far simpler. Details of system configurations and evaluation results can be found in [26].

V. RELATED WORK

Component-based modeling, design, and integration [28] have been the subject of study for many years in the area of software engineering, but only recently they are used for control applications in the manufacturing domain to support reusability and reconfigurability. Most of these software engineering techniques have been developed to aid system design, and components are implemented thereafter. With these techniques, one can only achieve only design model reusability and reconfigurability. There are some architectures like CORBA, DCOM, Java Bean, and Jini that support implementation reuse and reconfiguration, but they heavily depend on middleware services, which are not suitable for machine-level software construction due to their high system overhead and weak real-time support. Moreover, the nonreconfigurable behaviors of components, which are normally hard coded, limit the reusability and reconfigurability with the presence of some improved middleware system (such as real-time CORBA), since the machine control application requires behavior reconfiguration more frequently than classical computer applications, such as internet and scientific computation.

Agha *et al.* developed an architecture based on the Actor model for real-time system design [3], [21]. In such an architecture, the software consists of a set of Actors that can be preimplemented code and communications among Actors. Each Actor has a thread dedicated to it. However, the behaviors of these implemented Actors are not reconfigurable. Selic *et al.*used a similar Actor concept in ROOM by adding structure, hierarchy,

[6]The MotionController component executes as an individual task with a period of 10 ms.

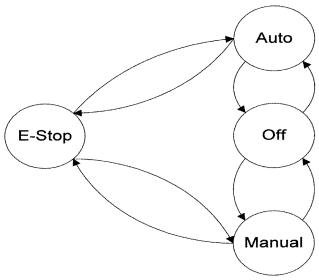


Fig. 19. Subsystem FSM.

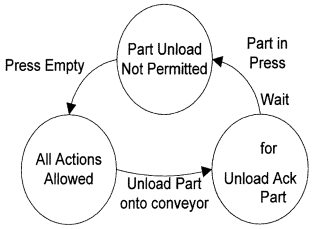


Fig. 20. Manual mode FSM.

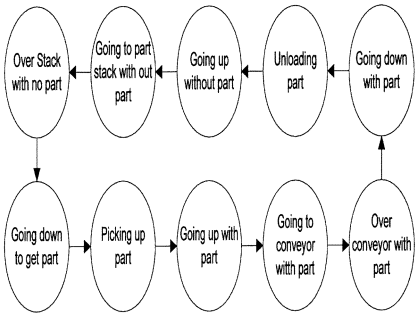


Fig. 21. Automatic mode FSM.

and behaviors to make it more suitable for real-time control applications [23]. All these models are developed mainly for design-phase abstractions and with a very limited consideration of reconfigurability after implementation. Although many open architecture controller researchers [8], [15] dealt with the reconfiguration problem, reconfigurability is supported only at the hardware level. For software reconfigurability, Stewart *et al.* developed an architecture in *Chimera* using port-based objects to support dynamic reconfiguration [25]. Such an architecture is not widely adopted since it is based on global shared memory which is not well supported in current distributed control systems. The Open System Architecture for Controls within Automation Systems (OSACA) [20] was implemented an open platform to support source-code level structural reconfiguration. Such reconfigurability is based on a set of OSACA communication services and requires high programming skills to regenerate the code when reconfiguration is required. OSACA does not address the application-level component modeling except for the interfaces, and does not support platform-independent component behavioral reconfiguration and reuse.

Since machine control systems are normally time critical, the behavior specification of a system needs to be formally verified. The formal methods used for machine control system specification and verification include Concurrent Sequential Protocol (CSP) [22], Colored PetriNet [9], StateChart [11], Net Condition Event Systems (NCES) [31], and ModeChart [14]. Although these formal methods can help in verifying the correctness of design and specification, they are not executable and errors can be introduced during the implementation of these specifications. Moreover, when the specifications are changed for a new application, the implemented behaviors cannot be reconfigured with these formal methods. Other methods attempt to use programming languages to specify behaviors [12], [16], [19]. Such approaches suffer verification difficulties. Ardis *et al.* showed that the formal methods are programming languages of behavioral specifications [2]. To combine the strength of both, Tsang and Lai [29] proposed a specification method for soft real-time systems based on Time-Estelle that can be both executable and verifiable. Barnett *et al.* [4] proposed an executable specification method based on Abstract State Machines (ASMs). Our approach is also based on the concept of executable specifications, but executability is extensively supported by a well-defined component structure at the executable code level.

Some standard efforts have also been made to support software reconfigurability. POSIX.4 [10] defined a set of services and interfaces at the operating system level to support transparent reconfiguration of a platform. Since different vendors implement these services and interfaces differently, true reconfigurability is not achieved. IEC 61 499 [13] (a follow-on version of IEC 1131) defined a set of function blocks with their behaviors in FSM for software modeling and construction of electrical control systems. The OMAC User Group developed standard interfaces of reusable components for control and manufacturing systems [17]. As one of research groups to prototype and evaluate OMAC solutions, we have developed a software architecture compliant with these standards, but provided better reconfigurability by introducing a control logic driver and the service protocol mechanisms in the component structure and executable behavioral specifications in the Control Plan.

The work presented here also extends our previous work on open-architecture controllers and modularized real-time controllers [5], [18], [24], [32], [33], [35].

VI. Conclusion and Future Work

In this paper, we presented a component-based architecture for constructing reconfigurable software for machine tool control systems. Such software reconfigurability will enable adaptation to enterprise-level requirement changes and support low-cost agile manufacturing. In this architecture, reconfigurable software consists of communicating components that are modeled with event-based external interfaces, a control logic driver, communication ports, and service protocols. The architecture separates the component functionality from its behavioral specifications and platform configuration. Behaviors of software in NFSMs are specified in the Control Plan separately from the component and system implementations, reused with components and loaded into the system at run time. Structural and behavioral reconfigurations can be achieved by changing the composition of components and modifying the Control Plan, respectively. Reconfiguration without structural changes inside the existing components does not require code regeneration, thereby achieving executable code level reconfigurability. Our evaluation on a machine tool motion controller showed that such software is more flexible, reusable, and reconfigurable.

Our future work will focus on design-time analyses of timing and resource constraints. These constraints are called *nonfunctional* constraints and are critically important to real-time controls. Usually, the timing constraints come from the requirements of control processes, which are closely related to the system behaviors. On the other hand, the resource constraints depend on the software components used for control, which are related to the functionality of the system. Since our architecture supports the separation of functional definitions from behavior specifications, these nonfunctional constraints can be analyzed individually, hence solving cross-cutting issues. Moreover, since behaviors are specified in the Control Plan program, timing and scheduling analyses can be done at a higher level, instead of at the implementation object code level.

Acknowledgment

The authors would like to thank O. Storoshchuk of General Motors for the industry application results in Section IV-C.

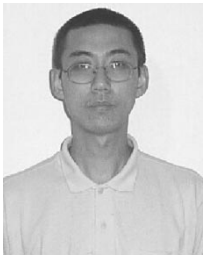
References

[1] R. Alur, S. Kannan, and M. Yannakakis, "Communicating hierarchical state machines," in *Proc. 26th Int. Colloquium on Automata Languages and Programming, Lecture Notes in Computer Science 1644*, 1999, pp. 169–178.

[2] M. A. Ardis *et al.*, "A framework for evaluating specification methods for reactive systems: Experience report," *IEEE Trans. Software Eng.*, vol. 22, pp. 378–389, June 1996.

[3] M. Astley and G. A. Agha, "A visualization model for concurrent systems," *Inform. Sci.*, vol. 93, no. 1–2, pp. 107–131, Aug. 1996.

[4] M. Barnett, E. Boerger, Y. Gurevich, W. Schulte, and M. Veanes. Using Abstract State Machines at Microsoft: A Case Study. [Online]. Available: http://reserach.microsoft.com/~gurevich /annotated.html
[5] S. Birla, "Software modeling for reconfigurable machine tool controllers," Ph.D. dissertation, Dept. Elec. Eng. Comp. Sci., Univ. Michigan, Ann Arbor, MI, 1997.
[6] "Requirements of Open, Modular Architecture Controllers for Applications in the Automotive Industry, Version 1.1," Chrysler, Ford, and GM, 1994.
[7] E. W. Endsley, M. R. Lucas, and D. M. Tilbury, "Software tools for verification of modular FSM based logic control for use in reconfigurable machining systems," in *Proc. 2000 Japan-USA Symp. Flexible Automation*, Ann Arbor, MI, July 23–26, 2000, pp. 565–568.
[8] "ESPRIT Consortium AMICS," in *CIMOSA: Open System Architecture for CIM*, 2nd ed. New York: Springer-Verlag, 1989.
[9] K. Feldmann *et al.*, "Specification, design and implementation of logic controllers based on colored petri net models and the standard IEC 1131," *IEEE Trans. Contr. Syst. Technol.*, vol. 7, pp. 657–674, Nov. 1999.
[10] B. O. Gallmeister, *Programming for the Real World: POSIX.4*. Sebastopol, CA: O'Reilly & Assoc., 1995.
[11] D. Harel, "On visual formalisms," *Commun. ACM*, vol. 31, no. 5, pp. 514–530, May 1998.
[12] J. Hooman and O. V. Roosmalen, "Timed-event abstraction and timing constraints in distributed real-time programming," in *Proc. 3rd Int. Workshop on Object-Oriented Real-Time Dependable Systems*, Newport Beach, CA, Feb. 5–7, 1997, pp. 153–170.
[13] "IEC 61 499—Function Blocks," International Electrotechnical Commission Technical Committee, 1999.
[14] F. Jahanian and A. K. Mok, "Modechart: A specification language for real-time systems," *IEEE Trans. Software Eng.*, vol. 20, pp. 933–947, Dec. 1994.
[15] Y. Koren, F. Jovane, and G. Pritschow, "Open Architecture Control Systems: Summary of Global Activity," ITIA—Institute for Industrial Technologies and Automation, ser. ITIA, vol. 2, 1998.
[16] K. Nilsen, "Java for real-time," *Real-Time Syst.*, vol. 11, no. 2, pp. 197–205, Sept. 1996.
[17] "OMAC API Documentation, Version 0.23," OMAC Working Group, http://www.isd.mel.nist.gov/project/omacapi/Bibliography/omacv023draft.pdf, Oct. 12, 1999.
[18] J. Park *et al.*, "An open architecture real-time controller for machining processes," in *Proc. 27th CRIP Int. Sem. Manufacturing Systems*, May 1995, pp. 27–34.
[19] C. Passerone *et al.*, "Modeling reactive system in java," in *Proc. 6th Int. Workshop on Hardware/Software Codesign (CODES/CASHE'98)*, Mar. 15–18, 1998, pp. 15–19.
[20] G. Pritschow and W. Sperling, "Modular system platform for open control systems," *Prod. Eng.*, vol. IV, no. 2, pp. 77–80, 1997.
[21] S. Ren and G. Agah, "A modular approach for programming distributed real-time systems," in *Lectures on Embedded Systems: Eur. Educational Forum School on Embedded Systems (LNCS 1494)*, Veldhovan, The Netherlands, Nov. 1996, pp. 171–207.
[22] S. Schneider, *Concurrent and Real-Time Systems: The CSP Approach*. New York: Wiley, 2000.
[23] B. Selic, G. Gullekson, and P. T. Ward, *Real-Time Object-Oriented Modeling*. New York: Wiley, 1994.
[24] C. Shiu, M. J. Washburn, S. Wang, C. V. Ravishankar, and K. G. Shin, "Specifying reconfigurable control flow for open architecture controllers," in *Proc.1998 Japan-USA Symp. on Flexible Automation*, vol. 2, Otsu, Japan, July 1998, pp. 659–666.
[25] D. B. Stewart, R. A. Volpe, and P. K. Khosla, "Design of dynamically reconfigurable real-time software using port-based objects," *IEEE Trans. Software Eng.*, vol. 23, pp. 759–775, Dec. 1997.
[26] O. Storoshchuk, S. Wang, and K. G. Shin, "Modeling manufacturing control software," in *Proc. 2001 IEEE Int. Conf. on Robotics and Automation*, Seoul, Korea, May 2001, pp. 4072–4077.
[27] (1999) Jini Architectural Overview: Technical White Paper. Sun Microsystems, Inc. [Online]. Available: http://www.sun.com/jini/whitepaper/architecture.html
[28] C. Szyperski, *Component Software: Beyond Object-Oriented Programming*. Reading, MA: Addison-Wesley, 1997.
[29] T. Tsang and R. Lai, "Specification and verification of multimedia synchronization scenarios using time-estelle," *Softw.-Pract. Exp.*, vol. 28, no. 11, pp. 1185–1211, Sept. 1998.
[30] T. Villa , T. Kam, R. K. Brayton, and A. Sangiovanni-Vincentelli, *Synthesis of Finite State Machines: Logic Optimization*. Norwell, MA: Kluwer, 1997.
[31] V. Vyatkin and H.-M. Hanisch, "A modeling approach for verification of IEC1499 function blocks using Net Condition/Event systems," in *Proc. IEEE Conf. Emerging Technologies in Factory Automation (ETFA'99)*, Magdeburg, Germany, Mar. 2000, pp. 72–79.
[32] S. Wang, C. V. Ravishanka, and K. G. Shin, "Open architecture controller software for integration of machine tool monitoring," in *Proc. 1999 IEEE Int. Conf. Robotics and Automation (ICRA'99)*, Detroit, MI, May 1999, pp. 1812–1820.
[33] S. Wang and K. G. Shin, "Generic programming paradigm for machine control," in *Proc. World Automation Congr. 2000*, Maui, HI, June 2000.
[34] A. M. Zaremski and J. M. Wing, "Specification matching of software components," *ACM Trans. Softw. Eng. Method.*, vol. 6, no. 4, pp. 333–369, Oct. 1997.
[35] L. Zhou *et al.*, "Performance evaluation of modular real-time controllers," in *Proc. ASME Dynam. Syst. Contr. Div., DSC*, vol. 58, Atlanta, GA, Nov. 1996, pp. 299–306.



Shige Wang (S'00) received the B.E. degree in computer engineering and the M.S. degree from Northeastern University, Shenyang, China, in 1989 and 1995, respectively. He is currently working toward the Ph.D. degree at the University of Michigan, Ann Arbor.

His current research interests are in embedded and control software models and architecture, software performance measurement, modeling, and analysis, and performance-aware software design and integration.

Kang G. Shin (S'75–M'78–SM'83–F'92) received the B.S. degree in electronics engineering from Seoul National University, Seoul, Korea, in 1970, and the M.S. and Ph.D. degrees in electrical engineering from Cornell University, Ithaca, NY, in 1976 and 1978, respectively.

From 1978 to 1982, he was with Rensselaer Polytechnic Institute, Troy, NY, as a faculty member. He has held Visiting positions at the US Air Force Flight Dynamics Laboratory; AT&T Bell Laboratories, NJ; the Department of Electrical Engineering and Computer Science, University of California, Berkeley; International Computer Science Institute, Berkeley, CA; the IBM T. J. Watson Research Center, Yorktown Heights, NY; and the Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA. He is currently a Professor of Computer Science and the Founding Director of the Real-Time Computing Laboratory, Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor. His current research interests are in QoS-sensitive networking and computing, embedded real-time OS, middleware, and applications.

Dr. Shin is a Fellow of ACM and a member of the Korean Academy of Engineering. He served as the General Chair of the 2000 IEEE Real-Time Technology and Applications Symposium, the Program Chair of the 1986 IEEE Real-Time Systems Symposium (RTSS), the General Chair of the 1987 RTSS, a Program Cochair of the 1992 International Conference on Parallel Processing, and has served on numerous technical program committees. He also chaired the IEEE Technical Committee on Real-Time Systems from 1991–1993 and was a Distinguished Visitor of the Computer Society of the IEEE. He was Guest Editor of the August, 1987 special issue of IEEE TRANSACTIONS ON COMPUTERS on real-time systems, an Editor of the IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED COMPUTING, and an Area Editor of the *International Journal of Time-Critical Computing Systems*, *Computer Networks*, and *ACM Transactions on Embedded Systems*.