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

Reconfigurable Software for Open Architecture Controllers *

Shige Wang and Kang G. Shin
Real-Time Computing Laboratory
Department of Electrical Engineering and Computer Science
The University of Michigan
1301 Beal Avenue
Ann Arbor, MI 48109-2122
email: {wangsg,kgshin}@eecs.umich.edu

Abstract

Reconfigurable software is highly desired for open architecture machine control systems when new products require new machine configurations and applications. In this paper, we present an architecture that supports reconfigurable software construction. In this architecture, the controller software consists of a set of well-defined components and a set of behavior specifications. Each component is modeled with event-based external interfaces, a control logic driver and service protocols. Behaviors of components and their integration are modeled as Finite State Machines (FSMs) and Nested Finite State Machines (NFSMs). The behaviors of software can be specified in *Control Plan* programs and executed by control logic drivers at runtime. Such software supports executable code-level reconfiguration as well as remote behavioral reconfiguration. Reconfiguration with heterogeneous implementations and vendor-neutral products is also supported. Our evaluation of the motion control software for a milling machine controller demonstrated that the software constrcuted with this architecture has high reconfigurability and low development and maintenance costs.

Key Words: open architecture controllers, real-time systems, reconfigurable software, software architecture.

1 Introduction

Open architecture machine controllers enable integration of new requirements and functions, such as plugging in new devices, adding new control algorithms and replacing existing subsystems, in a modular manner [1], hence necessitating reconfigurable software to support rapid system construction, plug-and-play devices and easy maintenance [2,3]. Typically, software for a machine controller consists of various device drivers and control algorithms which usually exist as software components and are desired to be reused accross multiple applications. Software needs to be reconfigured when there is a need for changing devices, swapping control algorithms and modifying operations. Therefore, reconfigurable software will enable fast and low-cost system construction when there is a need for reconfiguration of platform or application.

However, the current control software is not reconfigurable due to proprietary designs, ad-hoc implementations and platform-dependent configurations, hence resulting in long production cycle and high maintenance costs. Although the concept of component-based software [4] has been shown to be useful for reconfigurable software development, certain architectural issues are still unclear in this domain. Components are still implemented with hard-coded information dedicated to some application, and hence cannot be reconfigured once the system is set up. There is also a lack of architectural mechanisms to support separation of configurations, implementations and operations by which the functionalities and behaviors of software can be developed and reconfigured.

Most existing software models [5-7] are used for design-phase abstraction of a system with very limited consideration of post-implementation changes, thereby making software reconfiguration difficult. On the other hand, most open architecture controller researchers [1,8] focus on reconfiguration at hardware-level without software reconfiguration. Some recent standard efforts, like IEC 61499 [9] and OMAC API [10], attempt to achieve software reconfigurability by defining component and composition models with uni-

*This work is supported in part by the Defense Advanced Research Agency (DARPA) under the US Airforce Contract No. F33615-00-C-1706.

fied interfaces, but are limited to design-phase structure. An architecture that supports reconfigrable real-time software using port-based objects was proposed in [11], but is difficult to apply for manufacturing control systems due to the global shared memory assumption in this architecture which is not generally supported in distributed machine control systems. OS-ACA [12] is another open architecture, but doesn't address the software reconfiguration issues directly.

In this paper, we present an software architecture for building reconfigurable software with reusable components. Our goal is to provide executable-code-level reconfigurability for control software. In this architecture, components are modeled with a set of external interfaces, communication ports, a control logic driver and service protocols. Components can be structurally composed by linking their communication ports to form a new component, and then mapped to a platform by customizing their service protocols. The behavior of each component executed by the control logic driver is modeled as a *Nested Finite State Machine* (NFSM), which is a formal model for compositional behaviors. NFSM supports incremental and formal behavior analyses. The behavior can be specified in a *Control Plan* program for runtime configuration locally or remotely. *Control Plan* is a language based on the NFSM model to specify control logic and operation sequences for both individual compenents and their integration. Our architecture with these models separates function definitions from behavior specifications, and enables software reconfiguration at executable-code-level. The architecture also separates other non-functional constraints, especially timing and resource constraints, from functionality and behavior integration, so that these constraints can be analyzed and verified incrementally and as early as at design phase.

The rest of this paper is organized as follows. Section 2 describes the architecture for reconfigurable software construction, including component structure and composition model, and structural reconfiguration. Section 3 presents the NFSM model, specifications in *Control Plan* and behavior reconfiguration. Section 4 evaluats the proposed architecture based on control software construction for a machine control system. The paper concludes with Section 5.

2 System Architecture

Reconfigurable software can be viewed as consisting of a set of inter-communicating components, each of which is a pre-implemented software module and used as a building block. Figure 1 shows the reconfigurable software structure of a control system.

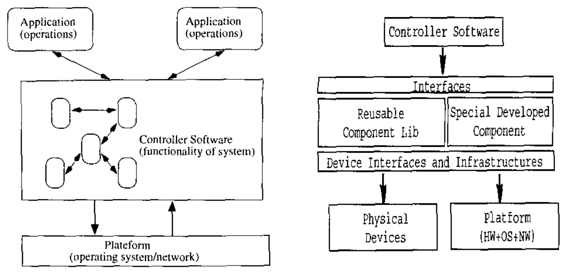

Application (operations)
Application (operations)
Controller Software (functionality of system)
Plateform (operating system/network)
Controller Software
Interfaces
Reusable Component Lib
Special Developed Component
Device Interfaces and Infrastructures
Physical Devices
Platform (HW+OS+NW)

Figure 1: Reconfigurable software structure.

2.1 Component structure

A software component consists of a set of external interfaces with registration and mapping mechanisms, communication ports, a control logic driver and service protocols, as shown in Figure 2.

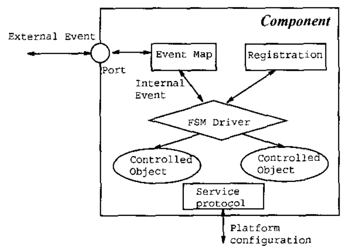


Figure 2: Reusable component structure.

External interfaces. External interfaces define functionalities of a component, i.e., operations that can be invoked. In our architecture, external interfaces are represented as a set of acceptable global (external) events with designated parameters. A customizable event mapping mechanism is devised in a component to achieve the translation between global events and the component's internal representations. A registration mechanism is further equipped to perform runtime check on received events. Only those operations invoked by authorized and acceptable events can be executed. Event-based interfaces enable operations to be scheduled and ordered adaptively in distributed and parallel environments, and allow components integrated into the system at executable-code-level.

Communication ports. Communication ports are used to connect components, i.e., physical interfaces of a component. Each reusable component can have one or more communication ports. The number of ports for 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 share one port.

Control logic driver. The control logic driver, also called the FSM driver, is designed to separate function definitions from control logic specifications, and support control logic reconfiguration at executable-code-level. The FSM driver can be viewed as an interface to access and modify the control logic inside a component, which is traditionally hard-coded in a component implementation. Every component involved in behavioral control should have such a driver inside. Control logic of a component can then be fully specified in ***state table*** for FSM driver executions. A FSM driver will generate commands to invoke operations of the controlled objects at runtime according to its state table and received events. State tables can also be packed as data and passed to another component to reconfigure behavior remotely.

Service protocols. Service protocols define execution environments or infrastructures of a component. Service protocols include scheduling policies, inter-process communication mechanisms and network protocols. A component can be used in different environments by selecting different service protocols. Such selection is based on the available mechanisms of a platform and performance constraints (such as timing and resource constraints) of the system.

2.2 Composition model and structural reconfiguration

Components are organized hierarchically in our composition model to support reconfiguration with different granularities. A high-level component is composed of inter-communicating low-level components, as shown in Figure 3, with a high-level control logic driver, some communication ports and customizable service protocols.

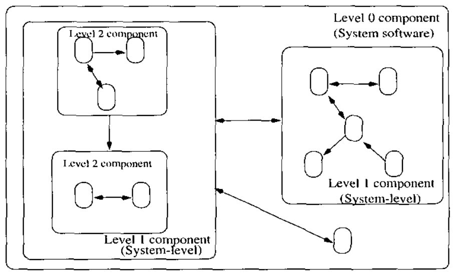


Figure 3: Heirarchical composition model

Structural reconfiguration is usually required when some functions are added, removed or replaced in the system. Such reconfiguration can be achieved by adding, removing or replacing the corresponding components at certain levels. Reconfiguration is also required when the existing components need to be reorganized to achieve a different function and/or when the platform is reconfigured. Such reconfiguration can be achieved by reorganizing connections among the components and/or customizing the service protocol when the runtime system is constructed.

3 Behavior Specification and Reconfiguration

Control applications are normally time- and safety-critical, and require software behaviors to be analyzed thoroughly before implementation. With the control logic driver that separates behavior specifications from function definitions in our architecture, the software behavior can be specified and verified separately from implemenations, then stored with components and loaded when the system starts. The behavior can also be reconfigured after implementation. Different component implementations can be selected and configured with the same behavior to satisfy different non-functional constraints.

3.1 Behavior specification

The behavior specifications of control software are divided into 2 disjoint parts: control logic specifications and operation sequence specifications.

Control logic specifications. Control logic specifications define the static part of software behavior or the control logic of a component. It is modeled as a NFSM with a set of traditional 'flat' FSMs organized hierarchically. A NFSM at level i, M_i, can be defined as:

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

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, and s_{i_0} is the initial state of M_i. A non-initial 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. Only FSMs of top-level components in a composition are visible during behavior configuration and verification. 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 state table with each entry defining a possible transition. The structure of each entry is:

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

where STATE is the current state of the system, $\mathsf{EVENT}_{\mathsf{input}}$ is an input event, ACTION_LIST specifies the actions to take or the functions to call, and $\mathsf{STATE}_{\mathsf{next}}$ is the component state after the transition. STATE and $\mathsf{EVENT}_{\mathsf{input}}$ together determine an entry in a state table uniquely, and consequently determine a unique set of operations and a unique next state.

Operation specifications. Operation specifications define the desired runtime input sequence that will trigger designated sequence of operations when no other interferences are involved. An operation specification can be specified as a pre-programmed event sequence consisting of a list of rows, each of which is with the format of:

[WHEN *state*] [INPUT e_{input} [PARAM *parameter*]]
OUTPUT e_{output} [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 portable and reusable reasons, 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 Figure 4:

```
FSM {label} [location]
      StateTable-entry1
      StateTable-entry2
      ......
ENDFSM

OPERATION {label} [location}
      operation-element1
      operation-element2
      ......
ENDOPERATION
```

Figure 4: Structure of a control plan program.

A FSM and ENDFSM block specifies a FSM state table while an OPERATION and 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 one component for runtime reconfiguration. A block can also be attached to an event as data to pass around. Details can be found in [13].

Specifications in other models. Behavior specifications in other models or langauges can be converted to a control plan using translators. Translators are programs designed to deal with the problem of heterogeneous models and specifications in a system. Translators are domain-specific and specification language-dependent, meaning that each translator can only convert programs in a designate specification language to control plan. Therefore, several translators may be required in a system if there are programs in several different specification languages.

3.2 Behavioral reconfiguration

Behavioral reconfiguration includes control-logic changes and operation-sequence changes. 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. An operation sequence needs to be modified when the machine operation procedure changes (e.g., use the same machine to manufacture parts of another family). Such reconfiguration can be achieved by defining a new operation specification. Then, a new control plan with these configurations can be generated and loaded into the runtime system. Thus, new behaviors can be obtained without regenerating configurations and implementations of components.

Our architecture provides executable-code-level behavioral reconfigurability via the control logic driver mechanism. The control logic driver in each component enables the same component to execute different behaviors by loading different state tables and operation sequence specifications. The behavior specifications can be classified further as device-dependent behaviors 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 devices. The device-dependent behaviors are specific for a device or a configuration, and can be reused for different applications with the same device.

4 Evaluation

We evaluated the reconfigurability of a motion controller software constructed using the proposed architecture on a 3-axis milling machine. The primary function of this controller is to coordinate 3-axis motion with some designated algorithms. The software was running on 2 control boxes (with their own processors and memory) connected with peer-to-peer Ethernets.

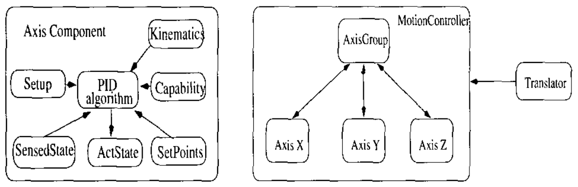
The software components were implemented with the structure and mechanisms described in the previous sections.

The components in this controller include control algorithms, physical device drivers and subsystems. Some high-level components in the motion controller and their functionalities are:

- **AxisGroup:** receives process models from the user or predefined control programs, and coordinates the motion of 3 axes by sending them the corresponding setpoints and desired velocities.
- **Axis:** receives commands from AxisGroup and sends out the signal to the controlled physical device according to the selected control algorithm (PID or FUZZY).
- **G-code Translator:** translates G-code program into control plan.

4.1 Structural reconfiguration

Figure 5 shows the structures of the original motion control software and axis component.



(a) Axis component.

(b) Motion controller.

Figure 5: Structures of motion control software and axis component.

The first reconfiguration was to replace the PID control algorithm in Axis component with a newly-developed Fuzzy control algorithm. This reconfiguration was achieved by replacing the PID algorithm component with the Fuzzy algorithm component inside the Axis component, as shown in Figure 6. This reconfiguration required code regeneration due to the functional changes inside the Axis component.

The second reconfiguration was to augment motion control with a newly-developed force supervisory control algorithm which adjusts the feedrate of axes dynamically according to the sensed forces. This reconfiguration was achieved by adding a new force supervisory component developed separately with a force sensor device driver into the system. In this reconfiguration, the new force supervisory component shared the same communication port with the one for AxisGroup commands. Since the AxisGroup component was implemented with the capability of changing feedrate with given values at the designated port, and the force supervisory algorithm was implemented outside the motion controller, this reconfiguration did not require code regeneration of the existing components and motion controller. The structure of motion control software after this reconfiguration is shown in Figure 7.

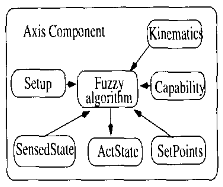


Figure 6: Axis with Fuzzy control.

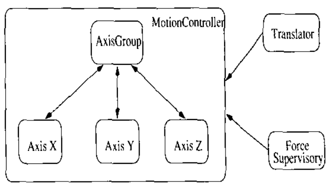


Figure 7: Motion control with force supervisory.

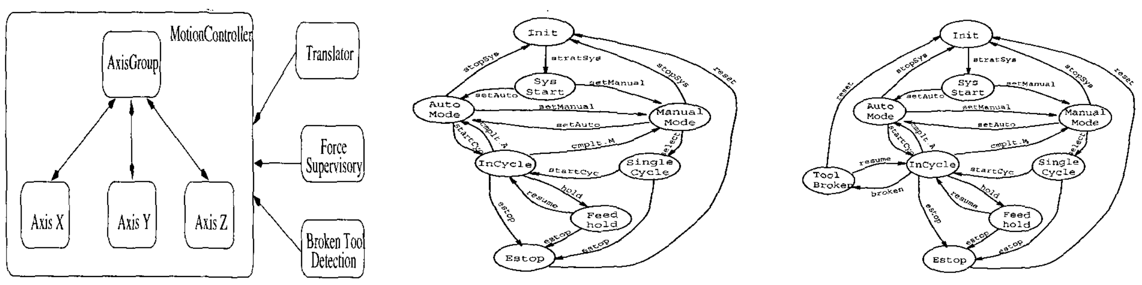
4.2 Behavioral reconfiguration

A broken tool detection algorithm was then developed and integrated into the motion controller to evaluate the behavioral reconfigurability of the software. The function of broken tool detection component is to detect abnormal forces at runtime, and immediately send a stop (*broken*) signal to the motion controller upon detection of such a force. The component was developed and integrated in the same way as the force supervisory control component. To react to the new broken tool signal, the machine-level control logic was changed to implement a new behavior upon receiving the broken tool signal, while the rest of behaviors remain unchanged. Figure 8 shows both structure and behavior changes for integration of broken tool detection. These reconfigurations did not require code regeneration of the existing software since the broken tool component was added outside the motion controller component and machine-level behavior reconfiguration in a new state table did not affect the implementation.

Compared to our previous implementations of the controller with traditional software approaches on the same testbed, the time taken to build a system with reconfigurable software was reduced by more than 50%.

5 Conclusion

In this paper, we presented a component-based architecture to support reconfigurable software con-



(a) Structure with broken tool.

(b) Machine-level FSM before reconfiguration.

(c) Machine-level FSM after reconfiguration.

Figure 8: Reconfiguration for broken tool integration.

struction for open architecture controllers. In this architecture, reconfigurable software consists of intercommunicating components that are modeled with event-based external interfaces, a control logic driver, communication ports, and service protocols. Behaviors of software can be modeled as NFSMs and operation sequences, specified in Control Plan separately from the component and system implementations, stored and reused with components, and loaded into the system at runtime. The structural and behavioral reconfigurations can be achieved by changing the composition of components and modifying Control Plan, respectively. Reconfiguration without structural changes inside 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.

References

[1] Y. Koren, F. Jovane, and G. Pritschow, *Open Architecture Control Systems: Summary of Global Activity, ITIA series, vol. 2*, ITIA - Institute for Industrial Technologies and Automation, 1998.

[2] Chrysler, Ford, and GM. *Requirements of open, modular architecture controllers for applications in the automitive industry, version 1.1*, December 1994.

[3] General Motors Powertrain Group. *Future Controls Software Requirements*, April 7 1999.

[4] C. Szyperski, *Component Software: Beyond Object-Oriented Programming*, Addison-Wesley Publishing Company, 1997.

[5] G. A. Agha and W. Kim, "Actors: A unifying model for parallel and distributed computing," *Journal of Systems Architecture*, vol. 45, no. 15, pp. 1263–1277, 1999.

[6] S. Schneider, *Concurrent And Real-Time Systems: The CSP approach*, John Wiley & Sons, Ltd., 2000.

[7] B. Selic, G. Gullekson, and P. T. Ward, *Real-Time Object-Oriented Modeling*, John Wiley & Sons, Inc., 1994.

[8] ESPRIT Consortium AMICS, *CIMOSA: Open System Architecture for CIM, 2nd revised and extended edition*, Springer-Verlag, 1989.

[9] International Electrotechnical Commission Technical Committee. *IEC 61499 — Function blocks*, 1999.

[10] OMAC working group. *OMAC API Documentation, version 0.23*, April 1999.

[11] D. B. Stewart, R. A. Volpe, and P. K. Khosla, "Design of dynamically reconfigurable real-time software using port-based objects," *IEEE Transactions on Software Engineering*, vol. 23, no. 12, pp. 759–775, December 1997.

[12] G. Pritschow and W. Sperling, "Modular system platform for open control systems," *Production Engineering*, vol. IV/2, pp. 77–80, 1997.

[13] S. Wang and K. G. Shin, "Generic programming paradigm for machine control," in *Proceedings of the World Automation Congress 2000*, MAUI, HA, June 2000.