An Architecture for Embedded Software Integration Using Reusable Components [*]

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

{wangsg,kgshin}@eecs.umich.edu

ABSTRACT

The state-of-art approaches to embedded real-time software development are very costly. The high development cost can be reduced significantly by using model-based integration of reusable components (models and/or software modules). To this end, we propose an architecture that supports integration of software components and their behaviors, and reconfiguration of component behavior at executable-code-level. In this architecture, components are designed and used as building blocks for integration, each of which is modeled with event-based external interfaces, a control logic driver, and service protocols. The behavior of each component is specified as a Finite State Machine (FSM), and the integrated behavior is modeled as a Nested Finite State Machine (NFSM). These behavior specifications can be packed into a *Control Plan* program, and loaded to a runtime system for execution or to a verification tool for analysis. With this architecture, embedded software can be constructed by selecting and then connecting (as needed) components in an asset library, specifying their behaviors and mapping them to an execution platform. Integration of heterogeneous implementations and vendor neutrality are also supported. Our evaluation based on machine tool control software development using this architecture has shown that it can reduce development and maintenance costs significantly, and provide high degrees of reusability and reconfigurability.

Keywords

embedded systems, reconfigurable software, software architecture, component-based integration.

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

1. INTRODUCTION

Agile and low-cost software development for real-time embedded systems has become critically important as embedded systems and devices are being used widely and customized frequently. However, the current practice in embedded software development relies heavily on ad-hoc implementation and labor-intensive tuning, verification and simulation to meet the various constraints of the underlying application, thereby incurring high development and maintenance costs. Although component-based software development and integration are known to be efficient for software development [29], such an approach is neither well-defined nor well-understood in the embedded real-time systems domain. Typically, embedded systems software consists of various device drivers and control algorithms, which usually exist as software components and are preferred to be reused for similar applications. Unfortunately, these components may contain dedicated information for some physical processes, and hence, can not be reused only based on their functions. The physical process for a target domain, on the other hand, is relatively static and can be modeled through component behaviors. Thus, components can be designed with customizable behavior mechanisms so that they can be reused for different applications. Besides supporting reuse and reconfiguration, the architecture for embedded software should also support separation of the specification and verification of non-functional constraints from those of functions. Such separation is essential for high-level implementation-independent specification and verification of non-functional constraints such as timing and resource constraints [24, 14].

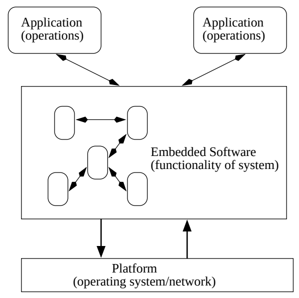
In this paper, we present an architecture that supports the above desired features for embedded software integration. Our reusable component model separates function definitions from behavior specifications, and enables behavior reconfiguration after structural composition. Components can be structurally integrated using their communication ports, through which acceptable external events can be exchanged to invoke target operations. The integrated software can then be mapped onto various platform configurations by customizing service protocols of components.

Behaviors of integrated software in our architecture are modeled as *Nested Finite State Machines* (NFSMs). The NFSM model supports compositional behavior specifications. It further supports incremental and formal behavior analysis. The behavior correctness of such an integrated system can be verified using an approach similar to that in [2]. Furthermore, since a given behavior can be implemented by different FSMs [30], different components may be selected for integration to meet different constraints while achieving the same behavior. The behaviors specified in other models or languages can be converted to this model using translators. The integrated behaviors can then be specified in a *Control Plan* program for remote and runtime behavior reconfiguration. Our 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.

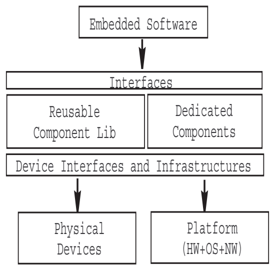
The rest of this paper is organized as follows. Section 2 describes the reusable component model. Section 3 presents the behavior model and the *Control Plan* for integrated behavior specification. Section 4 describes system integration under this architecture. Section 5 presents evaluations based on two example machine control systems built with the proposed architecture. Section 6 describes related work in this domain. The paper concludes with Section 7.

2. COMPONENT STRUCTURE

Components are pre-implemented software modules and treated as building blocks in integration. The integrated embedded software can be viewed as a collection of communicating reusable components. Figure 1 shows the embedded software constructed by integrating components.



(a) Embedded system structure



(b) Embedded software construction

Figure 1: Integration of embedded software components

The component structure defines the required information for components to cooperate with others in a system. Our software component is modeled as a set of external interfaces with registration and mapping mechanisms, communication ports, control logic driver and service protocols, as shown in Figure 2.

External interfaces. External interfaces define the functionality of the component that can be invoked outside the component. In our model, external interfaces are represented as a set of acceptable events with designated parameters. A component with other forms of external interfaces, such as function calls, can be integrated into the system by mapping each of them to a unique event. Using events as external interfaces enables operations to be scheduled and ordered adaptively in distributed and parallel en-

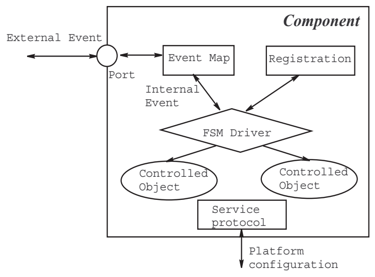


Figure 2: Reusable component structure.

vironments, and enables components from different vendors (possibly implemented with different considerations) to be integrated into the system without the source code.

External interfaces are generally defined as global (external) events, which are system-wide information. 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.

Communication ports. Communication ports are used to connect reusable components, i.e., they are physical interfaces of a component. Each reusable component can have one or more communication ports. The number of ports needed for a component can be determined and customized by the system integrator. Different types of ports with different service protocols can be selected to achieve different performance requirements. Multiple communication parties can share one port.

Finite State Machine 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. The FSM driver can be viewed as an internal interface to access and modify the control logic, which is traditionally hard-coded in software implementation. Every component that controls behaviors should have a FSM driver inside itself. Control logic of a component can now be modeled as a FSM and fully specified in a table form [30], or a *state table*. The FSM driver will then generate commands to invoke operations of the controlled objects at runtime according to the state table and the events received.

The FSM driver enables the control logic to be reused, and hence solves some cross-cutting design issues on physical process information blended with component behaviors. Typically, the behavior of a component is designed to control some physical process and is traditionally hard coded in the implementation of the component tailored to an execution environment. Therefore, the design of software and the design of an execution platform have to be done cooperatively. In our approach, since the behavior of a component is separated from its implementation by introducing the FSM driver, the design of sofware behavior and the platform configuration can be done independently and separately before integration.

The FSM driver and state tables also enable remote and runtime control logic reconfiguration. The state table can be treated simply as data and passed around the system. A state table can be partitioned into several small pieces with only one loaded to the FSM driver at a time. A component can also be reconfigured with a different state table when the external environment changes or upon other components' requests. This is even more useful for devices in a system with limited resources and unreliable environments, such as an in-vehicle control system.

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

3. BEHAVIOR MODEL AND SPECIFICATIONS

Embedded systems normally deal with mission- or safety-critical applications. Hence, the behavior of software should be thoroughly analyzed before its implementation. In our architecture, the behavior of integrated software is modeled as a NFSM while component behaviors are modeled as traditional FSMs. Both of them can be formally specified and verified.

3.1 Control Logic Specification

Control logic specifications are used to define the static behavior or the control logic of a component. Control logic for integrated software is modeled as a NFSM. A NFSM contains 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 i-th 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} 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 control logic of a component is modeled as a FSM (if it is a leaf node in the component hierarchy) or a NFSM (if it is a non-leaf node in the component hierarchy) and can be implemented independently. Only the top-level FSM of a component is visible during integration.

The FSM of a component can be fully specified in table form, where each entry defines a possible transition with the following structure:

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

where STATE is the current state of the system, $\mathsf{EVENT}_{\mathsf{input}}$ is the incoming event, ACTION_LIST specifies the actions to take or the functions to call, and $\mathsf{STATE}_{\mathsf{next}}$ is the next state that the component should reach after the transition.

All states and events have unique identifiers. STATE and $\mathsf{EVENT}_{\mathsf{input}}$ together uniquely determine an entry in a state table, and hence uniquely determine a list of actions and the next state.

3.2 Operation Specifications

Operation specifications define the desired runtime behavior and can be specified as a programmed operation sequence that will trigger the component actions when there are no other interferences. The operation specifications will generate predefined input events for a component FSM at runtime.

The operation specifications consist of a list of operation descriptions in the form 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. Fields in sequare brackets are optional. If a field is not specified, the value of the field will be ignored when executed. Such structure simplifies the specifications for pure state-triggered operations and event-triggered operations.

One and only one event can be specified in OUTPUT in each row to prevent ambiguous execution. e_{input} can be a combination of several incoming events as in the specification language of Cicero [15]. Similarly, *state* can also be some combination of several local states from different FSMs. *parameter* is treated as a data chunk in the specification. How to interpret and execute with these data is up to the receiver.

Events used in operation specifications should be global events for portability and reusability. However, operation specifications using internal events specific for a component can be integrated into the overall specifications as attached data of a global event. This enables runtime reconfiguration of operation sequences for a component. The process of such a global event results in execution of the new attached operation specifications in the component.

3.3 Specifications in Control Plan

A program that contains control logic and operation sequences for components is called a *Control Plan*. A control plan consists of two parts: *logic definitions* and *operation specifications*, corresponding to the control logic and operation specifications, respectively. The structure of a control plan is shown in Figure 3:

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

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

Figure 3: Structure of control plan program.

The block FSM and ENDFSM specifies the FSM state table for a component, while the block OPERATION and ENDOPERATION specifies the designed operation sequence. The label is used to identify blocks and components. The location is an option to specify where the block will be executed. A block will be executed at the current local site if the location is not specified in the OPERATION block.

It is possible for a control plan to have multiple FSM and OPERATION blocks for one component for reconfiguration. A block can also be attached to an event in either FSM or OPERATION block as data to pass around. More details of control plan specification and execution can be found in [32].

3.4 Behavior Specifications in Other Models

Since different subsystems of an embedded system may deal with different physical processes, it is possible that component behaviors are expressed in other models. Although the system integration of components in multiple models is an active research [28, 11] and the results are directly applicable to our architecture, we adopt translators to solve the problem of integrating heterogeneous models.

In our architecture, a translator is designed and implemented as a software component that converts a specification in a given formalism of a model to a control plan and NFSM. Translators are domain-specific and specification language-dependent, meaning that each translator can only convert programs in a designated specification language to control plan. Thus, several translators may be required in a system if there are programs written in several different specification languages.

4. SYSTEM INTEGRATION

Software integration includes component selection and binding, and control plan construction (both control logic and operation sequence). A runtime system can be generated by mapping the integrated software onto a platform.

4.1 Composition Model

The composition model defines how software can be integrated with given components. Since each reusable component is implemented with a set of external interfaces that uniquely define its functionality, components can be selected based on the match of their interfaces and design specifications. The integration of reusable components can be viewed as linking the components with their external interfaces.

Reusable components in integrated software are organized hierarchically to support integration with different granularities, as illustrated in Figure 4.

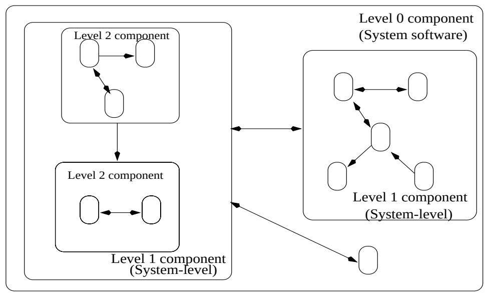


Figure 4: Hierarchical composition model

The behavior of an integrated component can then be modeled as integration of its member component behaviors. The control logic and operation sequences of each component can be determined individually and specified in a *Control Plan*. The behavior specifications can further be classified 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 dedicated to a device or a configuration, and can be reused for different applications with the same device.

With such a composition model, both components for low-level control such as algorithms and drivers and for high-level systems can be constructed and reused. However, additional overhead is introduced as the component level is increased, and may results in associated performance penalties due to excessive communications and code size.

4.2 Runtime System Construction

The integrated software obtained from the composition model cannot be executed directly on a platform since the composition model only deals with functionality. To obtain executable software, components have to be grouped into tasks, which are basic schedulable units in current operating systems.[1] Each task needs to be assigned to a processor with proper scheduling parameters (e.g., scheduling policy and priority) determined by an appropriate real-time analysis. Also, communications among components should be mapped to the services supported by the platform configuration. After these pieces of information are obtained, the components can be mapped to the platform by customizing their service protocols. Figure 5 shows the mapping from functional integrated software to a runtime system with our architecture.

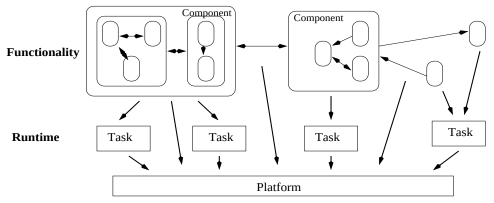


Figure 5: Runtime system generation from composition.

5. EVALUATION

We evaluated our architecture by constructing two software prototypes for machine tool control with the same set of reusable components. The software is running on two control boxes (with their own processors and memory) connected with a peer-to-peer Ethernet. The software components are implemented with the proposed structures and mechanisms. The evaluation is carried by examining the reusability of components and reconfigurability of integrated software.

5.1 Robotool Motion Controller

We first developed motion control software for a 3-axis milling machine, called Robotool. This controller is used to dynamically coordinate 3-axis motion with given algorithms and computed feedrate based on sensed forces.

The reusable components include control algorithms, physical device drivers and subsystems. Some high-level components used in the Robotool motion controller are:

- **AxisGroup:** receives a process model from the user or predefined control programs, and coordinates the

[1]Although there are some operating systems that support objects as basic units [9], they cannot be used for embedded applications currently due to their heavy overhead.

motion of the three axes by sending them the corresponding setpoints.

- **Axis:** receives setpoints from AxisGroup and sends out the drive signal to the physical device according to the selected control algorithm (PID or FUZZY).
- **G-code Translator:** translates a G-code program into a control plan.
- **Force Supervisory:** calculates the feedrate for AxisGroup at runtime.

Figure 6 shows the corresponding software structure.

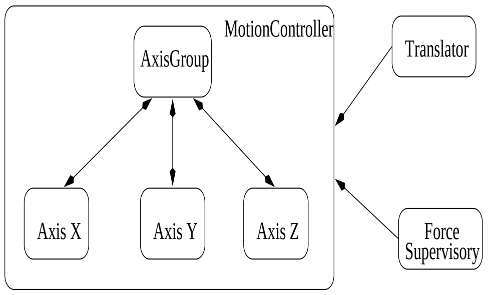
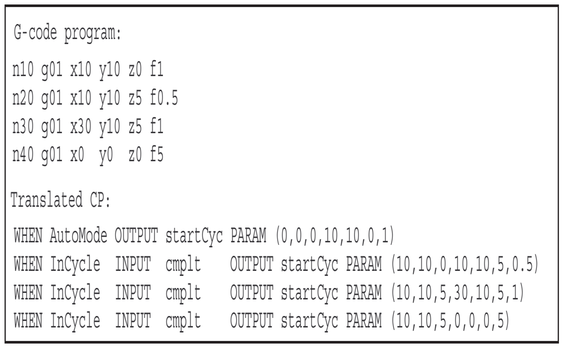


Figure 6: The structure of the Robotool motion control software.

The test application on the Robotool is a sequence of milling operations. The integrated behavior of the Robotool controller software consists of FSMs of Axis and AxisGroup components, and a G-code program for operation sequences. A higher-level motion control FSM is developed to specify the overall machine-level control logic. Figure 7 shows the control logic of each component and Figure 8 shows the desired operations.



```
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)
```

Figure 8: G-code and Control Plan.

Reconfiguration to include broken tool detection. A broken tool detection algorithm is then developed and integrated into the motion controller to evaluate the reconfigurability of the software. A broken tool detection algorithm is developed separately and implemented as an individual component. The function of the broken tool detection component is to detect abnormal forces at runtime, and send a stop signal to the motion controller when such a force is observed. The software structure with the broken tool detection algorithm is shown in Figure 9.

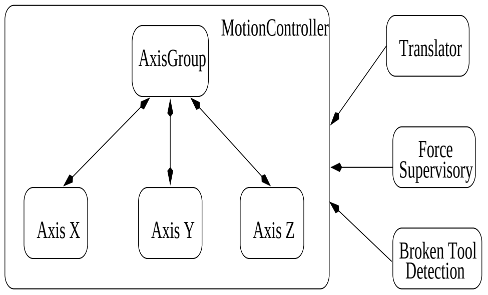


Figure 9: Software with broken tool detection.

To react to the new signal from the broken tool detection algorithm, the machine-level control logic needs to be changed, while the rest of behaviors remain the same. Figure 10 shows the new machine-level FSM.

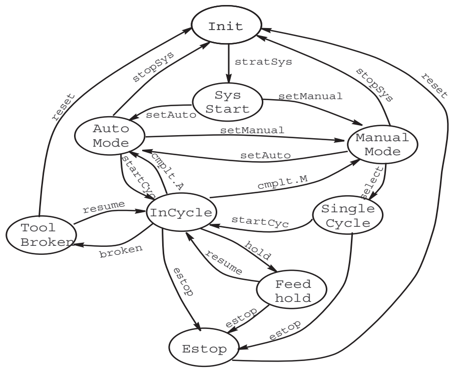


Figure 10: Machine-level FSM with broken tool detection.

5.2 Reconfigurable Machine Tool Controller

We then modified the Robotool motion control software to construct another motion control software for a Reconfigurable Machine Tool (RMT). RMT is a modularized and composable machine tool with 2-axis and a 2-position discrete device. Unlike the Robotool, the RMT motion controller needs neither coordinated motion nor monitoring.

The same axes and translator components are used to construct the RMT motion controller. A new component, *Spindle*, is added into the system to cnotrol the discrete device. The software structure for RMT controller is illustrated in Figure 11.

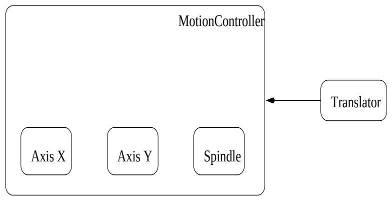
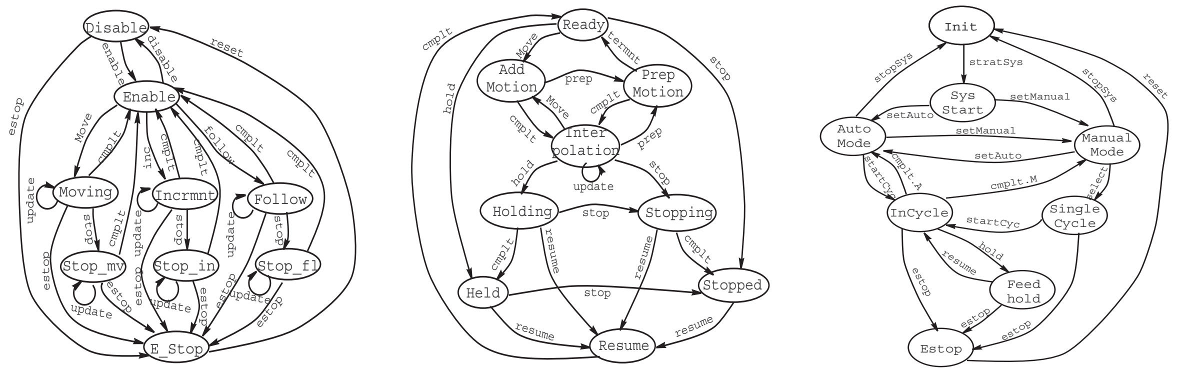


Figure 11: The structure of the RMT motion controller software.

The behavior specifications for Axis components are the same as those used for the Robotool. The behavior spec-



(a) Axis FSM (b) AxisGroup FSM. (c) Machine-level FSM.

Figure 7: Behavior specifications of Robotool motion controller.

ifications for the new added Spindle component is simple with only 3 states representing the position of *in* or *out* and the situation of *estop*, and transitions with corresponding events, as shown in Figure 12.

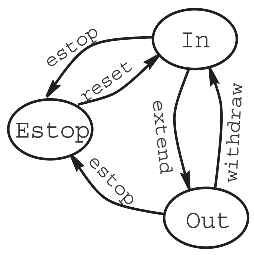


Figure 12: FSM for the spindle component.

Since the new Spindle component introduced new control logic into the system, the overall machine-level control logic had to be changed, and a new machine-level FSM was implemented, as shown in Figure 13. The G-code program

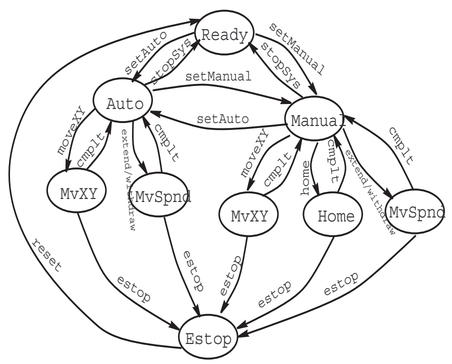


Figure 13: The machine-level FSM for the RMT.

and its corresponding control plan are also changed for new operation sequences as shown in Figure 14.

```
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)
```

Figure 14: G-code program and control plan for RMT.

5.3 Evaluation Results

According to our experience in implementing these controllers, the proposed architecture provides excellent reusability and reconfigurability of software. The component structure separates the functionality definition from the behavior specification so that the component can be reused for different applications. Furthermore, the behavior is separated from the component implementation, and therefore, components can be created and analyzed separately with the NFSM. Such an approach enables the system developer who is not familiar with the software, to do it on a daily basis. Separation of device-independent behaviors from device-dependent behaviors further increases the reusability of behaviors. As seen in the above applications, the Axis behavior can be reused when the component is reused in a different application.

Table 1 illustrates the numbers of components and behavior specifications in both controllers and the number of Robotool components and behaviors reused in RMT. [2] We were able to achieve over 80% reuse of components. On the other hand, the reuse of behavior specifications was not

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

at the same high percentage because a lower-level behavior change may require changes at higher-levels. However, since the behavior specifications and component functional specifications have been separated, the behavior changes will not affect the integration of components, and can be done without much of programming skills. These numbers are also consistent with the fact that functions are stable regardless of the application but the behaviors are not.

	Robotool	RMT	Reused
# of components	52	37	34
# of behavioral spec	6	5	2

Table 1: Number of components and behavioral specifications reused accross the controllers.

It also took less effort to reconfigure an existing controller or construct a new controller. As illustrated in Table 2, reconfiguring the Robotool controller software using our architecture with broken tool detection took much less effort than doing so to the software implemented using traditional approaches (0.8 human-month vs. 2 human-month). The time spent on the RMT software construction was also reduced by 50%.

	Robotool reconfiguration	RMT construction
Traditional approach	2	8
Proposed approach	0.8	4

Table 2: Efforts needed for two applications with traditional and proposed architectures (in human-month).

Although the proposed architecture demonstrated benefits in software construction, we experienced some performance penalties associated with it. Table 3 shows the execution times of the MotionController component,[3] collected by a special designed hardware, *VMEStopWatch card*, which has a built-in high resolution timer (25 nano-seconds). The overhead is possibly introduced by additional context switches for invoking components and event processing.

	Robotool	RMT
Traditional approach	2.1	1.3
Proposed approach	3.0	1.5

Table 3: Exectuion times for software constructed with different architectures (in millisecond).

6. RELATED WORK

Several models have been proposed in recent years to describe embedded software. Agha *et al.* developed an architecture based on the Actor model for embedded systems [24, 4, 1]. Selic *et al.* developed a Real-Time Object-Oriented Model that supports hierarchical actors and behavior integration [26]. An architecture based on the CSP model is also proposed to specify the embedded software [25]. Stewart, Volpe and Khosla developed an architecture for the *Chimera* project using port-based objects to support dynamic reconfigurable real-time software [27]. All of these architectures agree on modeling the components as autonomous self-contained software modules (called Actor or Process) and using event or message passing to describe the connection of components. However, the behaviors of a component in these architectures are not separated from its implementation, and therefore, it is difficult for engineers to reconfigure and analyze components and their integration. This also makes the performance analysis at design phase much more difficult since only abstract models are available. The performance analysis has to wait until these models are implemented, which will lengthen the development cycle due to the errors detected late in the cycle. Lack of context and environment descriptions may further introduce mismatch problems for both architecture and specification [33, 10] and interface inconsistency [6], especially when components from different sources are used.

Since many embedded systems deal with safety-critical applications, formal specification and verifiation are highly desired and widely used in software development. Various formal methods applicable in this domain are discussed and compared in [21]. The formal method used to describe the behaviors of integrated software should be composable. StateChart [12] is one of the methods widely used in current practice for behavior modeling. Jahanian and Mok proposed Modechart based on real-time logic to specify and analyze real-time systems [17]. Other methods, such as Net Condition Event Systems [31] and Colored PetriNet [8], are also introduced to specify and analyze the behaviors of integrated embedded software.

A disadvantage of using formal methods to specify a system is that an executable system cannot be constructed after specifications are done, and the implementation can introduce additional errors even if the specifications have been proved to be correct. Therefore, researchers suggested using programming languages, such as Ada and C/C++, to specify the behaviors directly [21]. Methods to specify real-time and reactive systems using Java have been proposed in [19, 22]. Tsang and Lai presented a method of using Time-Estelle to specify and verify soft real-time systems. Hooman and Roosmalen presented a method for extending programming language to specify composable and reusable real-time systems [13]. However, the correctness verification of all these approaches is still based on simulations. Ardis *et al.* evaluated different specification methods, and showed that the formal methods can satisfy more criteria [3].

Another issue for embedded software integration is model heterogeneity. Since a component of embedded software may deal with some physical process of the external world, its behaviors may be represented with the model which is most suitable for the physical process. Solutions for this problem include the multi-graph architecture developed at Vanderbilt University [28], the meta-architecture based on the Actor model at University of Illinois [1] and $*$*charts* at UC Berkeley [11].

The industry has also put significant efforts into the development of a practical architecture for embedded software integration. UML has been used for embedded software modeling [7]. The CORBA architecture has also been adapted in embedded system software domain [5]. The OMAC user group defined standard interfaces of reusable components for

[3]The MotionController component was executing as an individual task with period of 10 millisecond.

control and manufacturing systems [20]. The International Electrotechnical Commission Technical Committee proposed IEC 61499 function blocks to model and construct software for electrical control systems [16, 31].

Comparing with all the related work mentioned above, our proposed architecture supports all the desired features of embedded software integration, including multiple-granularity composability, executable-code-level reconfigurability, separation of functional, behavioral and timing specifications and incremental integration and verification. Our architecture is also compliant with industrial standards in current practices, and thus, can be applied to real-world applications in industry.

7. CONCLUSIONS AND FUTURE WORK

In this paper, we presented a component-based architecture for embedded software integration. This architecture defines components and a composition model as well as a behavior model. A reusable component in our architecture is modeled with a set of events as external interfaces, communication ports for connections, a control logic driver (FSM driver) for separate behavior specification and reconfiguration, and service protocols for executing environment adaptation. Such a structure enables multi-granularity and vendor-neutral component integration, as well as behavior reconfiguration.

The control logic of the integrated software is modeled using the formal method of NFSM, where each FSM represents the control logic of a component in the integration. The control logic of each component is specified in a state table separately from the component implementation, and can be reconfigured remotely and dynamically. Verification can also be done independently of implementation, and incrementally as the integration continues. A control plan is used to specify both control logic and operations. Control plan programs are portable and can be executed directly by a component. The behavior specified in a control plan can be further divided into device-dependent and device-independent parts, which can be reused for the device and different applications, respectively.

The evaluations based on two control applications has shown that the components with our proposed model improve the reusability and reconfigurability of integrated software, and significantly reduce development time and efforts.

Our future work will focus on the timing and resource analysis for integrated software. The architecture presented in this paper makes it possible to separate the timing and resource analyses from the functional integration and verification. To satisfy the timing constraints of integration software, we plan to apply deadline distribution [18] at every level of integration to obtain a feasible deadline for each component operation. Component-level scheduling algorithms will be developed for mapping components to tasks, and execution analysis methods [23] will also be used to analyze the resource requirements.

8. REFERENCES

[1] G. A. Agha and W. Kim. Actors: A unifying model for parallel and distributed computing. *Journal of Systems Architecture*, 45(15):1263–1277, 1999.

[2] R. Alur and M. Yannakakis. Model checking of hierarchical state machines. In *Proceedings of the 6th ACM Symposium on Foundations of Software Engineering*, pages 175–188, Lake Buena Vista, FL, 1998.

[3] M. A. Ardis *et al.* A framework for evaluating specification methods for reactive systems: experience report. *IEEE Transactions on Software Engineering*, 22(6):378–389, June 1996.

[4] M. Astley and G. A. Agha. A visualization model for concurrent systems. *Information Sciences*, 93(1-2):107–131, august 1996.

[5] K. Black. Corba corsses the embedded systems. White Paper.

[6] J. Bosch. Object acquaintance selection and binding. *Theory and Practice of Object Systems*, 4(3):151–168, 1998.

[7] B. P. Douglass. *Real-time UML: developing efficient objects for embedded systems.* Addison-Wesley, 2000.

[8] K. Feldmann *et al.* Specification, design, and implementation of logic controllers based on colored petri net models and the standard iec 1131. *IEEE Transactions on Control Systems Technology*, 7(6):657–674, November 1999.

[9] D. L. Galli. *Distributed Operating Systems: Concepts and Practice.* Prentice Hall, Upper Saddle River, NJ, 2000.

[10] D. Garlan, R. Allen, and J. Ockerbloom. Architecture mismatch: why reuse is so hard. *IEEE Software*, 12(6):17–26, november 1995.

[11] A. Girault, B. Lee, and E. A. Lee. Hierarchical finite state machines with multiple concurrency models. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 18(6):742–760, 1999.

[12] D. Harel. On visual formalisms. *Communications of the ACM*, 31(5):514–530, May 1998.

[13] J. Hooman and O. V. Roosmalen. Timed-event abstraction and timing constraints in distributed real-time programming. In *Proceedings of the 3rd International Workshop on Object-Oriented Real-Time Dependable Systems*, pages 153–170, Newport Beach, CA., February 5-7 1997.

[14] J. Hooman and O. V. Roosmalen. An approach to platform independent real-time programming: (1)Formal description. *The International Journal of Time-Critical Computing Systems*, 19:61–85, 2000.

[15] Y. M. Huang and C. V. Ravishankar. Constructive protocol specification using cicero. *IEEE Transaction on Software Engineering*, 24(4):252–267, April 1998.

[16] International Electrotechnical Commission Technical Committee. IEC 61499 — function blocks, 1999.

[17] F. Jahanian and A. K. Mok. Modechart: A specification language for real-time systems. *IEEE Transactions on Software Engineering*, 20(12):933–947, December 1994.

[18] J. Jonsson and K. G. Shin. Deadline assignments in distributed hard real-time systems with relaxed locality constraints. In *Proceedings of the IEEE International Conference on Distributed Computing Systems*, pages 432–440, Baltimore, MD, May 27-30 1997.

[19] K. Nilsen. Java for real-time. *Real-Time Systems*, 11(2):197–205, September 1996.

[20] OMAC working group. Omac api documentation, version 0.23, April 1999.

[21] J. O. Ostroff. Formal methods for the specification and design of real-time safety critical systems. http://www.cs.yorku.ca/ jonathan/survey /combined-paper.html.

[22] C. Passerone *et al.* Modeling reactive system in java. In *Proceedings of the 6th International Workshop on Hardware/Software Codesign (CODES/CASHE'98)*, pages 15–19, March 15-18 1998.

[23] D. Peng and K. G. Shin. Modeling of concurrent task execution in a distributed system for real-time control. *IEEE Transactions on Computers*, C-36(4):500–516, April 1987.

[24] S. Ren and G. Agah. A modular approach for programming distributed real-time systems. In *Lectures on Embedded Systems: European Educational Forum School on Embedded Systems (LNCS 1494)*, pages 171–207. Springer-Verlag, Veldhovan, Netherland, november 1996.

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

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

[27] 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*, 23(12):759–775, December 1997.

[28] J. Sztipanovits *et al.* Multigraph: an architecture for model-integrated computing. In *Proceedings of the 1995 1st IEEE International Conference on Engineering of Complex Computer Systems*, pages 361–368, Ft.Lauderdale, FL, 1995.

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

[30] T. Villa *et al. Synthesis of Finite State Machines: logic Optimization*. Kluwer Academic Publishers, 1997.

[31] V. Vyatkin *et al.* Systematic modeling of discrete event systems with signal/event nets and its application to verification of iec 1499 function blocks. ftp://ftp.cle.ab.com/stds/iec/sc65bwg7tf3/html/news.htm, 1998.

[32] S. Wang and K. G. Shin. Generic programming paradigm for machine control. In *Proceedings of the World Automation Congress 2000*, June 2000.

[33] A. M. Zaremski and J. M. Wing. Specification matching of software components. *ACM Transactions on Software Engineering & Methodology*, 6(4):333–369, october 1997.