A Component-Based Model Integrated Framework for Embedded Software*

Wenzhi Chen, Cheng Xie, Jiaoying Shi

College of Computer Science, Zhejiang University, Hangzhou 312207, P.R.China
wzchen@cad.zju.edu.cn,
arthurxie@vip.sina.com, jyshi@cad.zju.edu.cn

Abstract. The development of distributed, concurrent software in embedded systems is becoming increasingly complex and error-prone. Model-based integration of reusable components is advocated as the method of choice. To this end, we propose a framework to support component-based model integration, hierarchical functionality composition, and reconfiguration of systems with continuous and discrete dynamics. In this framework, components are designed and used as building blocks for integration, each of which is modeled with abstract ports, reactions, and communication schemes. It uses hierarchical composition to hide the implementation details of components, and keeps the components at the same level of hierarchy interacting under a well-defined model of computation. Code generation takes the design decisions down to the final running system. Within this framework, embedded software can be constructed by selecting and then connecting components in a functionality repository, specifying models and transforming them to executable codes.

1 Introduction

The model-based approach has proven to be effective for fast and low-cost design and simulation of embedded systems, such as automotive systems. However, due to the lack of a common framework, the benefits of model-based approach are limited by the manual process of extracting information in one model for reuse in another. Furthermore, the current practice in embedded software development relies heavily on ad-hoc implementation to meet the various constraints of the underlying platform. Although component-based software development and integration are known to be efficient for software reusability, such an approach is neither well-defined nor well-understood in the embedded system domain.

* This work was supported in part by the Hi-Tech Research and Development Program of China (863 Program) under Component-based Embedded Operating System and Developing Environment (No.2004AA1Z2050), and Embedded Software Platform for Ethernet Switch (No. 2003AA1Z2160); In part by the Science and Technology Program of Zhejiang province under Novel Distributed and Real-time Embedded Software Platform (No. 2004C21059).

Z. Wu et al. (Eds.): ICESS 2004, LNCS 3605, pp. 563-569, 2005.
© Springer-Verlag Berlin Heidelberg 2005

Embedded systems are intrinsically heterogeneous. It consists of various device drivers and various control algorithms, which usually exist as software components. The physical processes to be controlled are usually continuous but the algorithms are implemented using discrete software components. There are hybrid models that match different parts of a system, for example, continuous time(CT) models for ordinary differential equations, finite state machine(FSM) models for plant operations, discrete event(DE) models for network communication, and synchronous data flow(SDF) models for signal processing. Although each individual model is relatively well-understood, it is difficult and complex to implement the integration of heterogeneous models.

An effective solution is to construct a common component-based framework and use it for model integration. In this paper, we present a framework that supports the component-based model integration and implementation process. The framework provides a component repository and hierarchical models, and can be used to specify software structure, distributed functionality, and system constraints. Function definitions of a component are separated from non-functional aspects, especially timing and resource constraints. Components can be structurally integrated via their communication ports, through which the state transitions of the system can trigger reactions. The functionality of a component can be implemented using a different model and enables reconfiguration after structural composition. The framework provides a clean way to integrate different models by hierarchically composing heterogeneous components. This hierarchical composition allows one to manage the complexity of a design by information hiding and component reuse. The framework has been applied to the Pcanel operating system that we developed at Zhejiang University for cybernetic transport system. The model integration framework allows seamless composition of vehicle applications with distributed real-time functionality to enforce desired efficiency and safety.

2 Component Repository

A component-based embedded software design is modeled as a set of software components and their interactions. Components are pre-defined software modules and treated as building blocks in integration. The integrated embedded software can be viewed as a collection of communicating reusable components.

The component repository contains the core software components for reusability and integrated descriptions about hardware and bus systems. The characteristics of the software components are also stored in the repository, e.g. test case, code size and worst case execution time. Interfaces must be part of the repository. In distributed embedded systems, the communication among software components can take place by data buses or internally on the processor.

The interfaces of the software components are defined globally. A formal notion of component interface provides a way to describe the interaction between components, and to verify the compatibility between components automatically. The theory of timed interfaces [1] is used to specify both the timing of the inputs a component expect from the environment, and the timing of the outputs it can produce. The

formalism of resource interfaces [4] is used to specify component interfaces that expose component requirements on limited resources.

The component structure defines the required information for components to cooperate with others in a system. Execution profiles define the execution environment or infrastructure of a component. Examples include scheduling policies, real-time constraints and resource demands. A component can be customized for use in different environments by selecting different execution profiles. Components have a collection of abstract input/output ports. Ports are shared states that allow components to communicate with each other via tokens. The number of ports needed for a component can be determined and customized by the system integrator. Different types of ports with different execution profiles can be selected to achieve different performance requirements.

Reactions define the functionality of the component that can be invoked outside the component. In our model, reactions are represented as a set of triggers with actions. Triggers are guards of some meaningful system states, such as time, signals, and events. A component with other forms of reactions, such as function calls, can be integrated into the system by mapping each of them to a unique trigger. Using triggers enables actions to be scheduled and ordered adaptively in distributed and concurrent system, and enables components from different vendors to be integrated into the system without the source code modification. With such a component model, the system can be designed by connecting cooperating components through their ports, and the system execution can be done by having external state transitions like timer interrupts or sensors trigger a sequence of reactions in components.

The semantics of reaction is designed to separate function definitions from state transition specifications, and support reconfiguration. Reactions of a component are specified in a table form [7]. When a trigger is activated at runtime, actions are invoked according to the state table. The table enables the control logic to be reused, and enables remote or runtime reconfiguration. The state table can be treated simply as data and passed around the system. This compactness of table is useful for embedded systems with limited resources and distributed environments, such as in-vehicle control systems. Figure 1 shows a component-based design for continuous time(CT) model and the component structure of corresponding implementation.

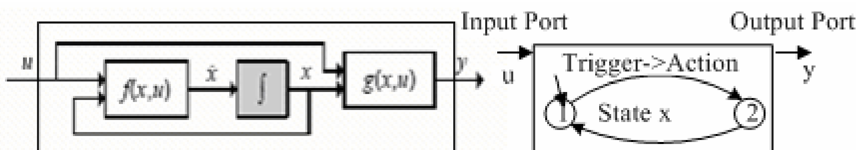

Fig. 1. Component-based design and implementation

Component connection network(CCN) combines the software components with each other. The framework supports hierarchical composition to keep the systemic view. The communication among components is carried out on token basis. The token flows are scheduled within models. In a hybrid system, hierarchical heterogeneous models cooperatively direct the token flows. Based on the CCN, token flow network is constructed to analysis and verify concurrent and real-time functionality of complete embedded software.

The complete software can not independently from the hardware. The execution of software depends on the underlying processor architecture, memory mapping, data bus, or device register. For reuse of components, hardware platform descriptions are also stored in the repository.

3 Distributed Functionality

More and more embedded systems consist of a network of electronic control units (ECU) connected via a bus. As the platform architecture shown in Fig. 2, each ECU consists of the controller, an operating system, a dedicated communication layer, and one or many application reactions. The functionality of a component is modeled as component structure. The functionality of a system is modeled as component connection network. The communication among components is modeled as token flow network. The distribution of functionality among ECUs is transparent in a high-level systemic view. The communication between reactions of spatially separated ECUs is wrapped by the communication layer. The integrated system model may span hybrid bus systems, such as Controller Area Network and Local Interconnect Network.

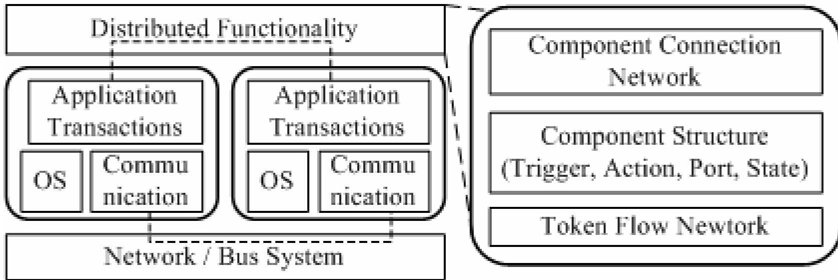

Fig. 2. Platform architecture

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

A composite is an integration of reusable components. The model of the composite links the components with their reactions, allowing for the observation and manipulation of the runtime states and behaviors internal of components. Furthermore, to facilitate modularity, a composite itself, together with the components within its model, can be treated as a integrated component at a higher level of hierarchy, which means that a composite can be encapsulated to a component. The member component behaviors determine the reactions and states of the integrated component. When applied formal models, the composite maintains assurance of diversified non-functional aspect, such as timing and deadlock.

Models are independent of implementation of components. Thus, Reusable components in integrated software are organized hierarchically to support integration with different models. A complete system configuration is a set of hierarchical compositions of models and reusable components.

4 Code Generation

The integrated software obtained from the composition model cannot be executed directly on a platform since the composition model only deals with distributed functionality. Code generation approach is a migration path from design-time models to runtime models. A typical code generation process assumes a flat operating system support and generates a stand-alone program that is then compiled into an application. Our framework provides a runtime system natively supporting executable models and distributed deployment. It greatly helps code generation and improves the quality of final software. The runtime system can utilize hardware support (such as SMP) and communication systems (such as CAN). In addition, there are certain assumptions, like resource reservation and timing predictability, can only be achieved by OS-level runtime systems, but not easily by stand-along programs.

To obtain and deploy complete software, components have to be transformed to reactions, which are basic schedulable units of the runtime infrastructure. A reaction is synthesized by code generator from a sequence of actions associated with an external trigger, which represents physical process such as interrupt, signal, and event. The code generator generates a runtime implementation that consists of a network of computing blocks communicating through a publisher-subscriber service.

A synthesized reaction has access rights on internal states of all components that own the actions. Such access rights are constructed during reaction initialization to avoid concurrent data competition. On arrival of a trigger, the reaction executes the pre-compiled actions in static order. The reaction is not reentrant, that in each round of a reaction execution, exactly one trigger is processed.

All reactions execute with statically assigned priorities in the runtime model. A reaction with high priority preempts lower-priority reactions. Actions within a reaction are executed at the same priority assigned to the reaction itself. The execution sequence of actions modeled at design-time to achieve functionality is preserved at runtime. Compared to other models requiring dynamic priority assignment [5], our implementation has low runtime overhead, lesser complexity and better support for massive concurrency.

5 Related Work

Since most embedded systems deal with safety-critical applications, model-based design and formal analysis are highly desired and widely used in software development. Ren et al developed an approach based on the Actor model for distributed real-time systems [6]. An Integrated Object-Oriented Environment is proposed for Real-Time Industrial Automation Systems [2]. Stewart et al used port-

based objects to support dynamic reconfigurable real-time software [3]. All of these frameworks agree on modeling the components as autonomous self-contained software modules and using event mechanisms to describe the connection of components. However, Most of the previous research in the literature has focused on component model, while largely ignoring the heterogeneous properties of software for hybrid systems. Systematically integrating heterogeneous components is crucial to design complex embedded systems. It is difficult for engineers to reconfigure and analyze components and their integration. Lack of context and environment descriptions may further introduce mismatching problems of architecture and interface inconsistency. Out framework is inspired by practical applications like automotive control applications which are model heterogeneity. The component-based model integrated framework we proposed is designed for re-usability of components with model heterogeneity.

6 Conclusion

In this paper, we presented a component-based model integrated framework for embedded software. A reusable component in our framework is modeled with communication ports, triggers, and reactions for separate functionality specification and reconfiguration. Component repository contains components for reusability and integrated descriptions for executing environment adaptation. Distributed functionality within hybrid models is designed by hierarchical composition of components. Code generator transforms the design to implementation by OS-level runtime support. Such a framework enables multi-granularity and vendor-neutral component integration, as well as functionality reconfiguration. Our future work will focus on the timing and resource analysis for integrated components. The framework presented in this paper makes it possible to separate the timing and resource analyses from the functional integration.

References

1. Arindam Chakrabarti, Luca de Alfaro, Thomas A. Henzinger, and Marielle Stoelinga. Resource interfaces. Proceedings of the Third International Conference on Embedded Software (EMSOFT), Lecture Notes in Computer Science, Springer-Verlag, 2003.
2. Becker, L. B., Gergeleit, M., Nett, E., Pereira, ., C. E., An Integrated Environment for the Complete Development Cycle of an Object-Oriented Distributed Real-Time System, 2nd IEEE International Symposium on Object-oriented Real-time distributed Computing (ISORC'99), Saint-Malo, France, pp. 165-171, May 1999.
3. D. Stewart and P. Khosla, "The chimera methodology: Designing dynamically reconfigurable and reusable real-time software using port-based objects," International Journal of Software Engineering and Knowledge Engineering, vol. 6, no. 2, pp. 249-277, 1996.
4. L. de Alfaro, T.A. Henzinger, and M.I.A. Stoelinga. Timed interfaces. In Embedded Software, Lect. Notes in Comp. Sci. 2491, pages 108-122. Springer, 2002.

5. M. Saksena, P. Karvelas, and Y. Wang. Automatic synthesis of multi-tasking implementations from real-time objectoriented models. International Symposium on Object-Oriented Real-Time Distributed Computing, March 2000.
6. 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.
7. T. Villa, T. Kam, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. Synthesis of FSMs: Logic Optimization. Kluwer Academic Publishers, 1997.