8

OBJECT-ORIENTED DESIGN PATTERN APPROACH TO SEAMLESS MODELING, SIMULATION AND IMPLEMENTATION OF DISTRIBUTED CONTROL SYSTEMS

Satoshi Kanai[1], Takeshi Kishinami[1], Toyoaki Tomura[2], Kiyoshi Uehiro[3], Kazuhiro Ibuka[3], and Susumu Yamamoto[3]

[1] *Dept. of Systems Engineeering, Hokkaido Univ., Japan*
[2] *Asahikawa National Colledge of Technology, Japan*
[3] *Motorola Japan Ltd., Japan*
e-mail: kanai@coin.eng.hokudai.ac.jp

Abstract: Distributed control systems (DCS) come into wide use in automation areas. In this paper, an object-oriented design pattern approach for modeling, simulation and implementation of the DCS is proposed. The proposed design patterns enable the uniform modeling of the static structures and dynamic behaviors of the DCS, the transformation of the models into simulation program, and the generation of the embedded codes. The Java-based modeler and simulator, and code generator were developed based on these patterns. Applications to the building automation and factory automation systems proved its effectiveness.

Key words: object-oriented modeling, design pattern, distributed control system, simulation, UML, LonWorks, FieldBus.

1. INTRODUCTION

Distributed Control Systems (DCSs) using open networks such as Fieldbus, CAN and LonWorks, are rapidly replacing traditional centralized control systems in a factory, process and building automation areas [1]. As shown in Figure 1, the DCS generally consists of many devices and an open network interconnecting them. A device consists of one control node and several device components (sensors, actuators), while a composite device consists of a group of devices. The DCS can make the system more scalable, and its building and

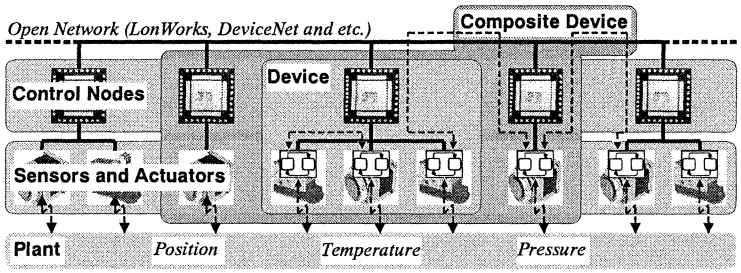

Figure 1. The structure of the DCS

wiring cost much less. On the other hand, the communication traffic among the devices tends to concentrate on a network. This causes an unreliable control performance. The system integrators have to test and avoid such situations after building the DCS. However, these works are time-consuming and costly.

To solve the problem, the system integrators need the computer-aided tools where they can simulate performances of the designed DCS, and can generate the embedded control codes installed into devices. The systematic method is needed to develop such tools. The method has to satisfy several requirements: 1) static structures of various devices and device components can be modeled, 2) dynamic behaviors in the devices and communication among the devices can be modeled, 3) the models can be easily used in the executable simulation program, and 4) the models can be automatically transformed to the embedded control codes installed in each device.

The object-oriented methods have been used for modeling various manufacturing systems. So far, SEMI/CIM-Framework[2], OSE[3], SEMI/OBEM[4] and GEM[5] specified the reference models of manufacturing resources described by UML or Coad&Yordon methods. However, their modeling scopes do not fit to the above requirements of the DCS.

The purpose of this research is to propose an object-oriented and design pattern–based method for seamless modeling, simulation and implementation of the DCS. Five design patterns are newly proposed. The DCS modeler and simulator, and the embedded code generator are also developed based on these patterns. A case study for controlling material handling system proves the effectiveness of our method and tools.

2. OVERVIEW OF PROPOSED DESIGN PATTERNS

Design pattern is a reusable structure for collaboration and interaction among classes or objects applied to capturing problems in a general domain [6]. However, existing design patterns are too generalized to express the DCS simulation models. Therefore, in this paper, five design patterns are proposed

specializing in modeling, simulating and implementing the DCS. Figure 2 shows

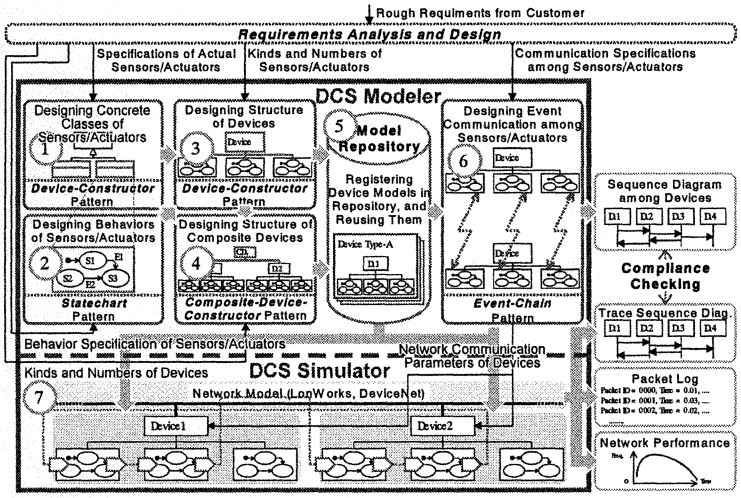

Figure 2. The design pattern-based modeling and simulating tools for the DCS

our modeling and simulating tools of the DCS and related design patterns.

Device-Constructor pattern describes the instantiation mechanisms for structuring the device models composed of many kinds of sensors, actuators, and local controllers. Composite-Device-Constructor pattern describes ones for structuring the composite device models composed of many kinds of devices. Statechart pattern defines the state-transition mechanism for realizing the dynamic behavior of each device and its device components. Event-Chain pattern defines the event dispatching mechanism among the sensors and actuators inside the device and inter-device on the network. Statechart-compiler pattern defines the mechanism for transforming the behavior of the DCS simulation model to the low-level codes embedded in the device. The first four patterns are designed for the DCS simulation, while the Statechart-compiler pattern is for the implementation. The details are described in section 3 and 4.

3. DESIGN PATTERNS FOR MODELING AND SIMULATION

3.1 Device and Composite Device Constructor Patterns

The Device-Constructor pattern and Composite-Device-Constructor pattern are proposed to model the static structure of the DCS[7]. Figure 3 shows the UML class diagram of the patterns. The Composite-Device-Constructor pattern

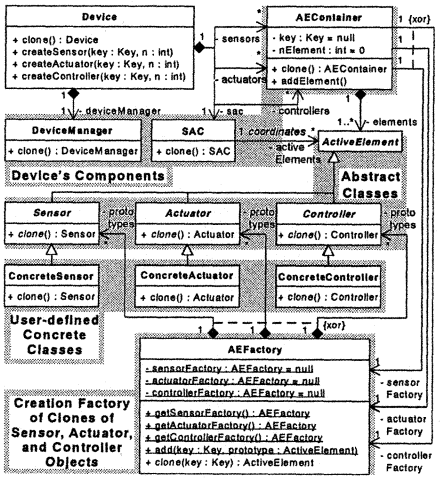

Figure 3. Device Constructor and Composite-Device-Constructor Patterns

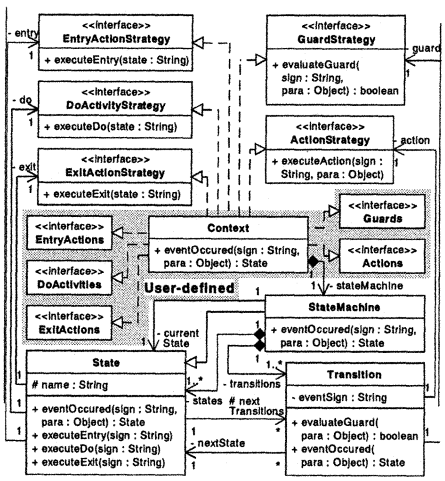

Figure 4. Statechart Pattern

is a pattern that extends three classes (*CompositeDevice*, *Device Container*, and *Device Factory*) of the Device Constructor pattern to the composite device. In this pattern, *AEContainer* represents the container class for all of the objects of a particular concrete class. *AEFactory* represents the factory class for creating the cloned objects of the concrete classes. The object of the *AEFactory* class contains the objects of each concrete class with a key string. *ConcreteSensor*, *Concrete Actuator*, and *Concrete Controller* represent the concrete classes of the three abstract classes *Sensor*, *Actuator* and *Controller*.

The features of this pattern are as follows. Firstly, a particular kind of device component can be modelled as a concrete subclass, so that we can directly represent the system structure that various types of sensors, actuators are connected to the control node. Secondly, the connection of a control node with its constituent device components can be expressed explicitly. Thirdly, various types of devices can be flexibly modelled only by changing the kind or the number of device components in the predefined device models. These features are similar to the ones of the Composite Device-Constructor pattern.

3.2 Statechart Pattern

The dynamic behavior of each device component in the DCS can be specified as the Statechart. The Statechart provides finite state machines with the notions of sub-states, entry and exit actions, do-activities and guards [8]. To accurately model these notions of the Statechart and realize the executable Java code, the Statechart patterns are newly proposed.

Figure 4 shows the class diagram of the Statechart pattern. *Context* has the dynamic behavior described by the Statechart. *StateMachine* describes a finite state machine composed of sets of states and transitions. *State* corresponds to either the state itself or its sub state machine. *EntryActions*, *DoActivities*, and *ExitActions* describe the actions and the activities in a state as interfaces. *Guards* and *Actions* represent guards and actions of a transition as interfaces. *EntryActionStrategy*, *DoActivityStrategy*, and *ExitAction Strategy* represent the concrete implementation methods to execute actions and activities. *GuardStrategy* and *ActionStrategy* also represent the ones to evaluate transition conditions and to execute actions according to occurred events.

In the proposed Statechart pattern, the one-to-one simple mapping from the design pattern to the Java code can be explicitly defined. Moreover, because the general structure of the Statechart itself and the implementation of actions, activities and transitions depending on the context can be completely separated in the pattern. So the system designer can easily identify and modify the methods of actions, activities and guards only in the Context's class

3.3 Event-chain Pattern

To complete event-driven DCS simulation models, the event links between device components transmitted both at the inter-device and intra-device levels have to be modeled and dispatched as shown in Figure 5. For modeling these event links, we propose the Event-chain Pattern.

Figure 6 shows the class diagram of an Event-Chain pattern newly proposed. *IOVariable* represents the class of input/output variables of the device components, device, and composite device. Any subclass of the *IOVariable* class can be defined freely according to the specification of network. *Link* represents the class of links between the input and output variables, including peer-to-peer, multicast, and broadcast communications. *EventDispatcher*

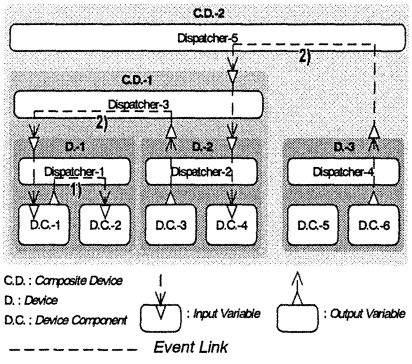

Figure 5. Event links and dispatchers

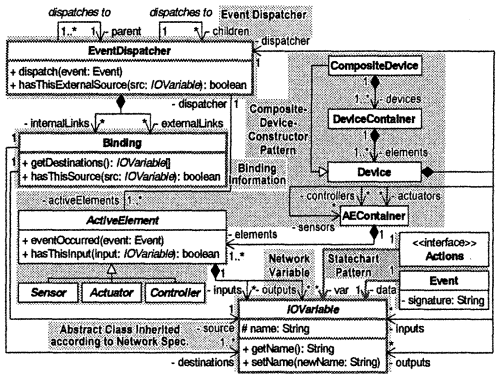

Figure 6. Event-chain Pattern

represents the class of event dispatchers that determine the destination of the generated events of the device.

In the Event-Chain pattern, the *Device* and *CompositeDevice* objects only have to relate to a uniform *EventDispatcher*, and they do not have to care if the event is going into or out from the device. The existing *Device* and *CompositeDevice* objects including pre-defined links are also reusable in the modelling of other *CompositeDevice* objects. The change of *IOVariable* objects' value is interpreted as an event or an action in the Statechart pattern of concrete device classes. Using the procedure, the DCS simulation model can automatically dispatch all the events among the suitable device components.

4. STATECHART-COMPILER PATTERN

The behavior of the control code executed in the device can be modeled as Statechart, but in case of LonWorks-based DCS, the code is eventually implemented as Neuron C. The Neuron C [9] is a subset of ANSI C, so that the Statechart behavior must be implemented without "class" concept. The Statechart-compiler pattern is proposed to bridge this gap. This design pattern can transform the device behavior modeled as Statechart to the Neuron-C code.

Figure 7 shows the process to apply the Statechart-Complier pattern. Firstly, the Statechart is converted to the textual formal description whose syntax is specified by the extended BNF[10]. Secondly, the formal description is parsed to obtain a syntax tree. Each node of the tree corresponds to the object defined in

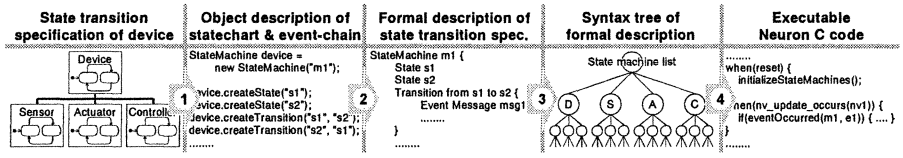

Figure 7. The process of control code implementation using Statechart-Compiler pattern

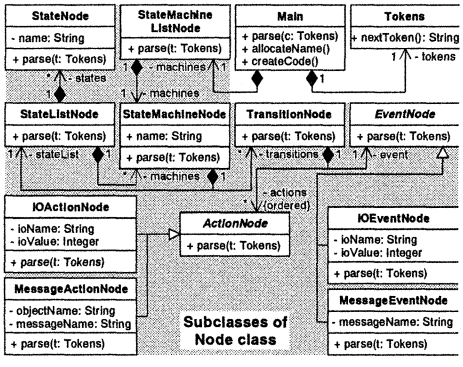

Figure 8. Statechart-Compiler pattern

Table 1. A mapping rule from Statechart-Compiler pattern to Neuron C

Event and Action			Neuron C code
Event	I/O event	(Start of event chain)	when(**io_changes**(*io_name*) to *value*) { if(**eventOccurred**(&*machine*, &*event*)) { /* Its action(s) */ } }
	Message event	From different device (Start of event chain)	when(**nv_update_occurs**(*message_name*)) { if(*message_name* == ST_ON) { if(**eventOccurred**(&*machine*, &*event*)) { /* Its action(s) */ } } }
		From same device	if(**eventOccurred**(&*machine*, &*event*)) { /* Its action(s) */ }
Action	I/O action		**Io_out**(*io_name*, *value*);
	Message action	To different device	*message_name* = *value*;
		To same device	*is equal to message event from same device.*

the Statechart-complier pattern. Finally, the Neuron C code can be automatically generated by the mapping rules from objects of the syntax tree to the Neuron C statements. Table 1 is a part of these rules.

Figure 8 shows a class diagram of the Statechart-Compiler pattern. *Main* executes a method of generating Neuron C code after reading the formal description. A *StateMachineNode* represents one state machine described in the formal description. A *StateNode* represents a state in the statechart, and *TransitionNode* does a inter-state transition. *EventNode* and *ActionNode* represent an event or an action defined in the transition. *IOEventNode* represents an incoming I/O signal event from a sensor or an actuator, while *IOActionNode* represents an outgoing signal event to them. *MessageEventNode* and *MessageActionNode* represent message events received from or sent to other state machines.

The syntax tree does not depend on any programming language. Therefore, this pattern is applicable to implementing the control codes by the languages besides Neuron C. The control code of the other language can be built by re-defining the language-specific mapping rules from the syntax tree to the statements of that language. The automatic generation of the embedded control code for DCS can be realized by this systematic procedure.

5. A CASE STUDY OF DCS DEVELOPMENT

We have implemented the Java-based DCS modeler and simulator software, and Neuron-C code generator based on our patterns and procedures. In the simulator, a designer can predict network traffic and a packet log shown in Figure 9. By applying our design patterns, the modeler and simulator development could be completed only for two months. These simulation tools were introduced to several system integrators of building automation, and their effectiveness on the DCS development has been verified[7].

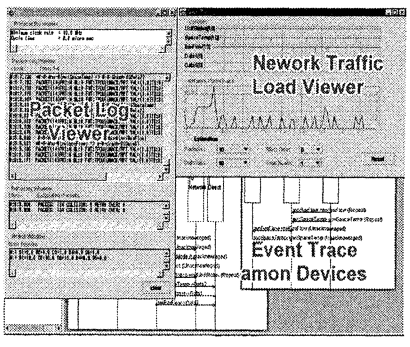

Figure 9. A screenshot of DCS simulator

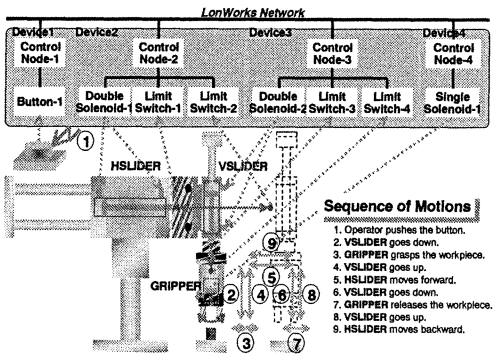

Figure 10. The DCS for pick & place unit

Moreover, as an application to the factory automation, we consider the DCS for controlling a pick-and-place unit driven by the pneumatic circuit. Figure 10 shows the structure of the DCS and the motion sequence of the unit. The DCS consists of LonWorks and four devices each of which consists of one control node, buttons, limit switches and valve actuators. Firstly, the task motions are described as a sequence diagram. Secondly, the Statechart of an object's behavior participating in a sequence diagram were identified. They were used with the Statechart and Event-Chain patterns. Using a Statechart-compiler pattern, four executable Neuron C codes have been generated and were installed into each of the four control nodes. As a result of the process, the motion sequence of the actual pick & place unit could be correctly worked by the DCS whose code was implemented from the simulation model.

6. CONCLUTIONS

In this paper, we proposed five object-oriented design patterns specializing in seamless modeling, simulation and implementation of the DCS. The DCS modeler, simulator, and embedded code generator could be efficiently developed based on the patterns. The result of using the modeler and simulator in system integrators, and the one of building the DCS for the material handling system controlled LonWorks proved the effectiveness of our approach.

REFERENCES

1. Dietrich, D., Neumann, P., and Schweinzer, H. (1999), *Fieldbus Technology – Systems Integration, Networking, and Engineering*, Springer-Verlag.
2. SEMATECH (1995), CIM Application Framework, Specification 1.2, 6.
3. OSE (1996), OSEC Architecture Version 2.0.
4. SEMI (1998): SEMI E98-0302 Provisional Standard for the Object-based Equipment Model (OBEM), SEMI.
5. SEMI(1998) : SEMI E30-0298 Generic Model for Communications and Control of SEMI Equipment (GEM), SEMI.
6. E. Gamma, R. Helm, R. Johnson, and J. Vlissides (1995), *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley.
7. Tomura, T., Kanai, S., Kishinami, T., Uehiro, K., and Yamamoto,S. (2001), Developing Simulation Models of Open Distributed Control System by Using Object-Oriented Structual and Behavioral Patterns", *Proceedings of Fourth IEEE Int. Symp.Object-oriented Real-time Distributed Compiting*, 428-437.
8. OMG (1999), Unified Modeling Language Specification Version 1.3.
9. Loy, D., Dietrich, D., and Schweinzer, H. (2001), *Open Control Networks: –Lon Works/EIA 709 Technology*, Kluwer Academic Publishers.
10. Aho, A. V., Sethi, R., and Ullman, J. D. (1985), *Compilers: Principles, Techniques, and Tools*, Addison Wesley.