Object-Oriented Design Pattern Approach for Modeling and Simulating Open Distributed Control System

Toyoaki Tomura  
Asahikawa National College of Technology, Japan  
e-mail: tomura@asahikawa-nct.ac.jp

Kiyoshi Uehiro  
Motorola Japan Ltd., Japan  
e-mail: rm175@email.sps.mot.com

Satoshi KANAI  
Hokkaido Univ. School of Engg.

Susumu Yamamoto  
Motorola Japan Ltd., Japan

Abstract

An open distributed control system (DCS), which consists of a large number of devices and a single open network interconnecting those devices, is now used in many automation areas. One critical problem of the DCS, however, is that because there is massive traffic on the network, the system integrator must carefully tune the network traffic after the construction of the system to assure its control performance. Hence a DCS simulator is strongly needed. In our study, we propose an object-oriented design pattern approach as a uniform, rapid and accurate method of modeling and simulating a DCS. Two special design patterns are proposed: Statechart Pattern which defines classes and the state-transition execution mechanism for realizing the dynamic behavior of devices, and Device-Constructor Pattern which defines classes and their instantiation mechanism for realizing the structure of devices composed of many kinds of sensors and actuators. The systematic procedures from those patterns to the executable Java code of the simulation model are also discussed. Furthermore, the effectiveness of our approach was investigated through actually developing an in-house Java-based DCS simulator.

1. Introduction

In recent years, the DCS based on open networks has been rapidly replacing traditional centralized control systems in factory, process and building automation. As shown in Figure 1, the DCS generally consists of a large number of devices composed of sensors, actuators and local controllers, and of a high-speed open network interconnecting those devices. Now we have ControlNet, Foundation Fieldbus, LonWorks, Device-Net, and etc. [1] as examples of such a high-speed open network. The DCS is capable of making the control system more scalable, and its construction and wiring cost much less. For these reasons, the DCS is expected to be used more widely in automation areas in the future.

One critical problem of the DCS is that the number of devices tends to be large and massive communication traffics among those devices concentrates on one single network. This causes unacceptable time-delay or fatal loss of data which assure reliable control performance. To avoid this problem, the system integrator must optimize the network traffic by carefully tuning the several communicative timing parameters of each device after the construction of the system. However, this task, requiring extra time and cost, becomes a bottleneck for rapid development of the DCS. Therefore, the system integrator strongly needs simulation techniques which enable the integrator to make uniform models of the structure and dynamic behavior of each device, to quickly implement the models as executable codes, and accurately estimate the dynamic behavior of the network traffic among the devices through combining those models.

In order to satisfy those requirements, object-oriented methodology has been used in modeling several distributed systems. SEMATECH has proposed CIM Application Framework [2] for semiconductor production. It specifies the reference model of the structure of manufacturing resource classes and association among them, and of the dynamic behavior of objects. OSE has proposed OSEC (Open System Environment for Controllers) [3] for open NC controller. It specifies the reference model of structure of function block classes of NC controller and association among them. SEMI’s Common Device Model [4] specifies a reference model of static structure and dynamic behavior of each device composed of various sensors, actuators and local controllers in UML [5]. Those approaches provide the system integrators with object-oriented reference models that can be used in several technical domains, and are very useful for uniform analysis and description of specification of each model to be used for DCS simulation. However, the approaches still have some problems which prevent us from efficiently building executable models necessary for DCS simulation in our study. The problems are as follows:
1) The reference models can only define the minimum required structure of classes and association among them, or state-transitions, and they do not have a detailed mechanism for implementing them. In addition, they fail to specify mechanisms for creating objects dynamically, for uniform management of the objects of various concrete subclasses inherited from abstract reference models, and for interacting the objects to actually execute the state-transition.

2) The concrete implementing procedure to bridge between the reference models and the executable codes of the simulation model is not explicitly given.

Recently, “Design Pattern” approaches [6] and [7] are used in the object-oriented design process. The design pattern approach specifies reusable mechanisms for collaboration and communication among classes or among objects to solve common object-oriented problems in a general domain. Generally, the approach classifies design patterns into three categories: patterns for creating objects, patterns for defining the structure of objects, and patterns for defining the behavior of the objects. Although the design pattern approach can solve the first problem mentioned above by combining several basic patterns, it still fails to solve the second one. In order to solve the two problems at the same time, we propose a new object-oriented design approach for modeling and simulating a DCS. The purposes of our study are:

1) We propose two design patterns for modeling a DCS, that is, Statechart which defines classes and state-transition execution mechanism for realizing the dynamic behavior of devices, and Device-Constructor which defines classes and their instantiation mechanism for realizing the static structure of devices.

2) We also discuss a systematic implementation procedure to bridge from those patterns to the executable Java code of the simulation model.

3) We confirm the effectiveness of our proposed approach based on those patterns through actually developing an in-house Java-based DCS simulator for building automation.

2. Modeling and Simulation of DCS

Figure 2 shows a process overview of modeling and simulation of the DCS based on the proposed two patterns. In each process, the system integrator’s tasks are as follows:

1) To construct the static structures and dynamic behaviors of the devices components by referring to the specifications of the components. The Device Constructor and Statechart patterns are used.

2) To construct the static structure of a device model by using the Device-Constructor pattern.

3) To connect the device model with the open DCS simulation model (which can simulate its communication protocol), and then evaluate network properties such as traffic on the network and packet collision frequencies.

In order to execute those processes, Motorola Japan, Inc. has developed a modeling tool for the device model and the DCS simulator using the proposed design patterns. This simulator is actually used in designing and evaluating the actual DCS governed by LonWorks for a heating-cooling system in the building automation area.

3. Statechart Pattern

In order to model a device, the system designer must develop all the device-specific component models with dynamic behavior. A statechart diagram [5] can describe the detailed dynamic behavior of a device-specific component model. In order to implement the statechart diagram systematically and easily, the system designer needs a design pattern and an implementation methodology based on it. In this section, we describe the Statechart pattern and the implementation methodology with the pattern, which are proposed in our study.

3.1 Statechart Diagram

The statechart diagram describes the dynamic behavior in common with all the objects of a class. In particular, it describes possible sequences of states and actions through which an object can proceed during its life as a result of reacting to discrete events. The notation of the statechart diagram are substantially those of Harel’s statechart [8] with extensions to make them object-oriented. Figure 3 shows the notation of the statechart diagram.
Table 1 Patterns for Implementing Statechart Diagram

<table>
<thead>
<tr>
<th>Pattern Elements</th>
<th>State</th>
<th>Transition</th>
<th>Event</th>
<th>Action and Activity</th>
<th>Current State</th>
<th>EntryConditions of Implementation</th>
<th>ExitConditions of Implementation</th>
<th>Support of Substates</th>
<th>Corresponding to Java Code</th>
<th>Performance of Execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>Case Statement in a Switch Statement</td>
<td>Changing Value of State Variable</td>
<td>Event Variable</td>
<td>Method of Context</td>
<td>State Variable</td>
<td>Good</td>
<td>Bad</td>
<td>Good</td>
<td>Excellent</td>
<td>Bad</td>
</tr>
<tr>
<td>Pattern</td>
<td>State</td>
<td>Transition</td>
<td>Event</td>
<td>Action and Activity</td>
<td>Current State</td>
<td>EntryConditions of Implementation</td>
<td>ExitConditions of Implementation</td>
<td>Support of Substates</td>
<td>Corresponding to Java Code</td>
<td>Performance of Execution</td>
</tr>
<tr>
<td>Pattern</td>
<td>State</td>
<td>Transition</td>
<td>Event</td>
<td>Action and Activity</td>
<td>Current State</td>
<td>EntryConditions of Implementation</td>
<td>ExitConditions of Implementation</td>
<td>Support of Substates</td>
<td>Corresponding to Java Code</td>
<td>Performance of Execution</td>
</tr>
<tr>
<td>Pattern</td>
<td>State</td>
<td>Transition</td>
<td>Event</td>
<td>Action and Activity</td>
<td>Current State</td>
<td>EntryConditions of Implementation</td>
<td>ExitConditions of Implementation</td>
<td>Support of Substates</td>
<td>Corresponding to Java Code</td>
<td>Performance of Execution</td>
</tr>
<tr>
<td>Pattern</td>
<td>State</td>
<td>Transition</td>
<td>Event</td>
<td>Action and Activity</td>
<td>Current State</td>
<td>EntryConditions of Implementation</td>
<td>ExitConditions of Implementation</td>
<td>Support of Substates</td>
<td>Corresponding to Java Code</td>
<td>Performance of Execution</td>
</tr>
<tr>
<td>Pattern</td>
<td>State</td>
<td>Transition</td>
<td>Event</td>
<td>Action and Activity</td>
<td>Current State</td>
<td>EntryConditions of Implementation</td>
<td>ExitConditions of Implementation</td>
<td>Support of Substates</td>
<td>Corresponding to Java Code</td>
<td>Performance of Execution</td>
</tr>
</tbody>
</table>

3.2 Previous Design Patterns

Table 1 shows design patterns for implementing the dynamic behavior described by statechart diagram [6], [7], [9]. These patterns except the Statechart pattern have critical problems in easiness of implementation, support of substates, and correspondance to the Java code. In order to solve these problems, the system designer needs a new design pattern which meets the following needs:

- The pattern has a mechanism for state-transition execution including transition in substates.
- The pattern can be easily implemented to the executable Java code by referring to a mapping rule.
- The implemented executable Java code can be easily modified by the system designer later.

3.3 Structure of Statechart Pattern

Figure 4 shows the structure of the Statechart pattern which is proposed in our study. The classes and interfaces in Figure 4 are defined as follows:

- **Context** is the class which has the dynamic behavior described by the statechart diagram. The object has one StateMachine object.
- **StateMachine** has two sets of states and transitions. The object is a either of statechart diagram, sequential substates, or one substate in concurrent substates.
- **EntryActions, DoActivities, and ExitActions** represent three sets of entry-actions, do-activities, and exit-actions respectively. Each action in a state is defined in these interfaces as an empty method.
- **Guards** and **Actions** represent two sets of guard-conditions and actions. Each condition and each action in transition are defined in these interfaces as an empty method.
- **EntryActionStrategy, DoActivityStrategy, and ExitActionStrategy** represent three logics in order to execute actions according to current states.

3.4 Implementation Methodology of Statechart Diagram with Statechart Pattern

Figure 5 shows the implementation methodology of statechart diagrams with the Statechart pattern. It consists of the following five processes:

1. Defining the five context-specific interfaces.
2. Adding all the methods of the defined interfaces to the class Context.
3. Describing all the relationships between state and transition.
4. Extracting the events and their parameters.
5. Transforming from the design model to the Java code by referring to the mapping rule.
Figure 6 An Example of Statechart Diagram

Table 2 Mapping Rule between Design Model and Java Code

<table>
<thead>
<tr>
<th>Statechart Diagram</th>
<th>Design Model</th>
<th>Java Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>State and Substates</td>
<td>Node in Tree Structure of States</td>
<td>State Object and StateMachine Object</td>
</tr>
<tr>
<td>Transition</td>
<td>Transition Relationship between States</td>
<td>Transition Object</td>
</tr>
<tr>
<td>Initial Pseudostate</td>
<td>Node in Tree Structure of States</td>
<td>Transition Object</td>
</tr>
<tr>
<td>Event and Its Parameter</td>
<td>Event String and Class of Parameters</td>
<td>Event String and Class of Parameters</td>
</tr>
<tr>
<td>Guard Condition</td>
<td>Empty Method of Interface Guards</td>
<td>Implemented Method of class Context</td>
</tr>
<tr>
<td>Action</td>
<td>Empty Method of Interfaces Actions, etc.</td>
<td>Implemented Method of class Context</td>
</tr>
</tbody>
</table>

As an example of implementation of a statechart diagram, we implement the statechart diagram shown in Figure 6 by applying the implementation methodology. In Figure 5, each process can be divided into the following steps:

1.1) Define the five context-specific interfaces.
1.2) Define each guard condition and each action as a method, and then add them to the interface Guards and Actions.
1.3) Define each action in all the states as a method, and then add them to the interfaces EntryActions, DoActivities, and ExitActions.
2.1) Add the methods of all the strategy interfaces to the context.
2.2) Add the methods of all the interfaces defined in the first process to the class Context.
3.1) Identify the tree structure of states, all the transition relationships between states, and all the initial pseudostates.
4.1) Extract event names from all the transitions.
4.2) If an event has one or more parameters, define a set of parameters as a class.
5.1) Transform from the design model to the Java code by referring to the mapping rule shown in Table 2.

Figure 7 shows the executable Java code of the statechart diagram shown in Figure 6.

4. Device-Constructor Pattern

In order to construct the static structure of a device model, we describe the Device-Constructor pattern and the construction methodology of the device model with it.

4.1 Previous Design Patterns

Figure 8 (a) shows the common device model. When the component class inherits any one of the three abstract classes, the configuration of the device components may become ambiguous. In order to solve this problem, an extended model can be considered as shown in Figure 8 (b). Because the class Device manages the component classes directly, the configuration of the components will be clear. However, the system designer must define the association between the class Device and the component class. The system designer must implement the procedure for creating the component objects into the class Device’s constructor. In order to solve these problems, the system designer needs a design pattern which meets the following requirements:

1) The pattern can define new component classes without associating between the class Device and them.
2) The pattern provides interfaces for creating the component objects without depending on the kinds of components and the numbers of their objects.
3) Interfaces provided by the pattern are very simple.
4.2 Structure of Device-Constructor Pattern

Figure 9 shows the structure of the Device-Constructor pattern, which is proposed in our study. In the Device-Constructor pattern, the classes are defined as follows:

- **Device** represents the device model defined by the SEMI Common Device Model.
- **AEContainer** represents the container for all the objects of a particular concrete class.
- **AEFactory** represents the factory for creating the cloned objects of the concrete classes.
- **ConcreteSensor**, **ConcreteActuator**, and **ConcreteController** represent the concrete classes of three abstract classes **Sensor**, **Actuator**, and **Controller**.

The Device-Constructor pattern partially extends the following recent design patterns: Abstract Factory, Factory Method, Prototype, Singleton, and Composite.

The Device-Constructor pattern meets the requirements described in section 4.1 by the following mechanisms:

1. The object of the class **AEFactory** contains the objects of each concrete class with a key string.
2. The object of the class **AEContainer** contains all the objects of a particular concrete class.
3. In order to create device components, the system designer must describe the sequence of class **Device** methods with the key string of the concrete class and the number of objects as two arguments.

4.3 Construction Methodology of Device Model with Device Constructor Pattern

Figure 10 shows the construction methodology of the device model with the Device-Constructor pattern. As an example of the device model, we consider a device model which consists of two sensors called **Sensor A** and **Sensor B**, and three actuators called **Actuator A**. The methodology consists of the following four processes:

1. By referring to the device components, define each component as a concrete class of the abstract classes **Sensor**, **Actuator**, and **Controller**, and then compile them. The dynamic behavior of the concrete class is implemented by using the Statechart pattern. Then allocate unique strings to each concrete class.
2. By referring to the defined concrete classes and their key strings, define the registration procedure of the concrete classes into the **AEFactory** object, as shown in Figure 11 (a).
3. In order to define the static structure of the device model, design the **AEContainer** objects which contain all the objects of a particular concrete class by drawing an object diagram as shown in Figure 12.
4. By referring to Figure 12, define the construction procedure of **Device** object shown in Figure 11 (b).
5. Modeling Tool and DCS Simulator

In order to construct device models in the DCS and simulate its dynamic behavior, Motorola Japan Inc. has developed the modeling tool of the device model and the DCS simulator. In the modeling tool, the Device-Constructor pattern is used for defining the static structure of the device model, and the Statechart pattern is used for implementing the dynamic behavior of the device model as the Java code. Figure 13 shows the snapshot of the modeling tool. In Figure 13, five kinds of device models are modeled, and are registered into the list of devices. By modeling the device models actually, we confirmed that two design patterns were very useful for rapidly developing the device models. Figure 14 shows the snapshot of the DCS simulator. The system designer can observe the performance of the network and the log of packets. Where the protocol of LonWorks is implemented as the network protocol of the DCS. By comparing the log of packets with the message sequence among the device models, we confirmed that two design patterns were very useful for simulating the dynamic behavior of the control network with the device models. By embedding two design patterns into the modeling tool and the DCS simulator, the period of time required for developing these applications was enormously shortened from four months to two months. Furthermore, because the newly defined component classes are imported into the DCS simulator alone, we consider that the efficiency of device modeling in the modeling tool is very high.

6. Conclusions

The conclusions of our study are summarized as follows:

1) We have proposed two design patterns for modeling a DCS. The Statechart pattern defines classes and the state-transition execution mechanism for realizing the dynamic behavior of devices. The Device-Constructor pattern defines classes and their instantiation mechanism for realizing the static structure of devices.

2) We have discussed the systematic implementation procedures to bridge from those patterns to the executable Java code of the simulation model.

3) We have confirmed the effectiveness of our approach based on these patterns through actually developing an in-house Java-based DCS simulator for the building automation.

References