UML4COP: UML-based DSML for Context-Aware Systems

Naoyasu Ubayashi  
Kyushu University  
ubayashi@acm.org

Yasutaka Kamei  
Kyushu University  
kamei@ait.kyushu-u.ac.jp

Abstract

Context-awareness plays an important role in developing flexible and adaptive systems. However, it is not easy to design and implement such a context-aware system, because its system configuration can be dynamically changed. This paper proposes UML4COP, a UML-based design method for COP (Context-Oriented Programming). UML4COP is a DSML (Domain-Specific Modeling Language) for designing context-aware systems. In UML4COP, each context is modeled separately and a system design model at a certain period of time is composed by merging associated contexts.

Categories and Subject Descriptors D.2.11 [Software Engineering]: Software Architectures—Languages

General Terms Design

Keywords Context-Aware System, Context-Oriented Programming

1. Introduction

Context-awareness improves the system usage and availability. Introducing the notion of context-awareness, we can develop flexible and adaptive systems that can change their behavior according to their context such as location [9]. However, it is not easy to design and implement such a context-aware system, because its system configuration can be dynamically changed. It is hard to check whether a design model is correctly implemented and its behavior is faithful to the design.

To deal with this problem, this paper applies the notion of COP (Context-Oriented Programming) [6] to a design method for developing context-aware systems. COP, a new programming paradigm, can treat context as a software module and enables programmers to describe the context-aware behavior elegantly. Using COP, context-dependent behavior can be separately described from the primary system behavior i.e., context-independent behavior.

This paper proposes UML4COP, a UML-based design method for COP. UML, (Unified Modeling Language) [12] is a standard modeling notation widely used in industries. UML4COP, a DSML (Domain-Specific Modeling Language) for designing context-aware systems, is a lightweight UML extension consisting of stereotypes specific to context-awareness. The semantics of standard UML diagrams are slightly changed to represent context-awareness. In UML4COP, each context is modeled separately from a base design model representing only primary system behavior. A system design model at a certain period of time is composed by merging associated contexts. Since a system design model contains multiple views including structural and behavioral aspects, it is preferable to independently model these views as contexts in terms of MDSOC (Multi-Dimensional Separation Of Concerns) [10]. Our approach is basically language-independent. That is, UML4COP can be applied to different COP languages.

The remainder of this paper is structured as follows. In Section 2, we introduce COP briefly. In Section 3, we show UML4COP. In Section 4, program implementation based on UML4COP is provided. In Section 5, we discuss the remaining problems. Concluding remarks are provided in Section 6.

2. COP

COP provides a mechanism for dynamically adapting the behavior to the new context. There are several COP languages such as ContextJ*, ContextJ, JCop (Java-based languages), and ContextL (Lisp-based languages) [1–3, 5, 6]. Using these COP languages, the primary system behavior can be separated from the context-aware behavior.

In most COP languages, context is described by layers, a context-aware modularization mechanism. A layer, which defines a set of related context-dependent behavioral variations, can be considered a software module.

By entering a layer or exiting from the layer, a program changes its behavior. That is, a program captures context-dependent behavior by entering a layer. A layer, a kind of crosscutting concern, can range over several classes and contain partial method definitions implementing behavioral variations. A set of partial methods belonging to the same layer represents the context-dependent behavior. There are two kinds of partial methods: plain method and layered method. The former is a method whose execution is not affected by layers. The latter consists of a base method definition, which is executed when no active layer provides a corresponding partial method, and partial method definitions. Partial methods are activated when a program enters a layer.
Layers are composed at run-time. The configuration of layers changes dynamically.
In COP, context-aware systems can be constructed by dynamically composing a set of associated layers, which are modules encapsulating the context-dependent behavior.

3. **UML4COP**

In this section, we propose UML4COP, which can represent context-dependent behavior in a modular way as illustrated in Figure 1.
3.1 Overview

Currently, design methods for COP are not yet proposed although several COP languages are provided as mentioned above. In this section, we introduce UML4COP in which context-dependent behavior is specified using standard UML notations and stereotypes specific to context-awareness.

UML4COP consists of two kinds of models: view model and context transition model. The former described in class diagrams and sequence diagrams represents context. The latter described in state machine diagrams represents context transitions triggered by COP-specific events such as layer in (entering a layer) and layer out (exiting from a layer).

3.2 View Model

Figure 1 shows an example model described in UML4COP. This example modified from [5] is an application that displays a message containing a person’s name, address, and employer. The message content changes according to the belonging context.

In UML4COP, a system design model is composed by multiple views representing base or layers. Each view model consists of class diagrams (structural aspects) and sequence diagrams (behavioral aspects). A base view represents the structure and behavior in case of belonging to no layer. A layer view represents the structures and behavior specific to a context. Plain methods and layered methods are defined in a base view and layer views, respectively. In Figure 1, there is one base view and two layer views: Address and Employment. In Address layer, a layered method toString is called to display an address. On the other hand, in Employment layer, another layered method toString is called to display an employer’s profile.

We can easily understand system behavior by composing views according to context transitions. Figure 2 shows two cases: 1) Address layer and 2) both Address and Employment layers. We can understand the behavioral difference between two cases. The output below shows an execution result of a program implementing the design model shown in Figure 1 and Figure 2. The same print statement of a person “Tanaka” behaves differently according to the context.

```
-- In Address Layer
Name: Tanaka; Address: Kyoto
-- In Address Layer and Employment Layer
Name: Tanaka; Address: Kyoto;
Employer Name: Suzuki; Address: Tokyo
```

3.3 Context Transition Model

In UML4COP, context transitions are specified using state machine diagrams as shown in Figure 3. Each state represents base or layer. In Figure 3, first, this example system
can enter Address layer. Next, the system can enter Employment layer (in this case, the system belongs to both of Address layer and Employment layer) or exit from Address layer. Figure 3 shows that the system cannot belong to only Employment layer. The order of entering a layer is also specified. To deal with the modeling complexity, it is possible to decompose a context transition model into hierarchically composed models if the system size is large.

As mentioned in this section, we can design a context-aware system in a modular way by introducing UML4COP in which both contexts and context transitions can be explicitly specified based on the notion of MDSOC.

4. Program Implementation Based on UML4COP

A design model in UML4COP can be easily implemented using COP languages. In this paper, we use ContextJ*. Although JCop is the most recent Java-based COP implementation, we use the old ContextJ* whose language features are provided as Java classes—new syntax is not introduced in ContextJ*.

In List 1 - 4 (Figure 4), we show a ContextJ* program implementing the design in Figure 1. In this program, two objects employer (suzuki) (List 1: line 03 - 04, List 4) and person (tanaka) (List 1: line 05 - 06, List 3) change their behavior corresponding to the context. Address and Employ-
Table 1. ContextJ® Execution Trace

<table>
<thead>
<tr>
<th>No.</th>
<th>Execution Event (ContextJ*)</th>
<th>Information</th>
<th>ContextJ® Code (Line)</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>[Layer with]</td>
<td>Address</td>
<td>List 1: line 08</td>
</tr>
<tr>
<td>02</td>
<td>[Method call]</td>
<td>println</td>
<td>List 1: line 10</td>
</tr>
<tr>
<td>03</td>
<td>[Method execution]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>04</td>
<td>[Method call]</td>
<td>toString</td>
<td>List 1: line 10</td>
</tr>
<tr>
<td>05</td>
<td>[Method execution]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>06</td>
<td>[Layered method call]</td>
<td>toString (Person’s Address layer)</td>
<td>List 3: line 35 - 38</td>
</tr>
<tr>
<td>07</td>
<td>[Layered method execution]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>08</td>
<td>[Base method call]</td>
<td>toString</td>
<td>List 3: line 20 - 22</td>
</tr>
<tr>
<td>09</td>
<td>[Base method execution]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>[Layer without]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>[Layer with]</td>
<td>Address</td>
<td>List 1: line 14 - 15</td>
</tr>
<tr>
<td>12</td>
<td>[Layer with]</td>
<td>Employment</td>
<td>List 1: line 14 - 15</td>
</tr>
<tr>
<td>13</td>
<td>[Method call]</td>
<td>println</td>
<td>List 1: line 17</td>
</tr>
<tr>
<td>14</td>
<td>[Method execution]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>[Method call]</td>
<td>toString</td>
<td>List 1: line 17</td>
</tr>
<tr>
<td>16</td>
<td>[Method execution]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>[Layered method execution]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>[Layered method call]</td>
<td>toString (Person’s Address layer)</td>
<td>List 3: line 35 - 38</td>
</tr>
<tr>
<td>20</td>
<td>[Layered method execution]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>[Base method call]</td>
<td>toString</td>
<td>List 3: line 20 - 22</td>
</tr>
<tr>
<td>22</td>
<td>[Base method execution]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>[Method call]</td>
<td>toString</td>
<td>List 3: line 37</td>
</tr>
<tr>
<td>24</td>
<td>[Method execution]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>[Layered method call]</td>
<td>toString (Employer’s Address layer)</td>
<td>List 4: line 24 - 27</td>
</tr>
<tr>
<td>26</td>
<td>[Layered method execution]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>27</td>
<td>[Base method call]</td>
<td>toString</td>
<td>List 4: line 17 - 19</td>
</tr>
<tr>
<td>28</td>
<td>[Base method execution]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>29</td>
<td>[Layer without]</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*layers are described in List 2. In ContextJ®, an object can enter a context by using with. For example, suzuki and tanaka enter Address and Employment layers (List 1: line 14 -15) and exit from the layers (List 1: line 19). The context views are mapped to layer descriptions ranging over two classes.

In COP, there are two kinds of layer declaration strategies [1]: class-in-layer and layer-in-class. The former is a strategy in which a layer is defined outside a class. This type is similar to aspect-orientation [7, 8]. A set of related layer definitions, a kind of crosscutting concerns, are completely separated from class definitions, a kind of primary concerns. On the other hand, layer-in-class is a strategy in which a layer is defined within a class. In this case, it is easy to understand the whole of a class definition. Each strategy has merits and demerits. In COP, a developer can choose either of them although ContextJ® supports only layer-in-class. UML4COP can deal with both strategies.

5. Discussion and Future Work

Although UML4COP and COP improve the expressiveness for designing and implementing context-aware systems, the essential problems specific to context-awareness still remain even if we use UML4COP and COP. In this section, we point out the problems in verifying context-aware systems.

Table 1 shows an actual logging trace of the example ContextJ® program.

Although the ContextJ® program is easy to read, the actual behavior is complicated. It is not necessarily easy to check whether this program correctly implements the design shown in Figure 1. Actually, this program behavior does not conform to the design. When tanaka (person) enters the Address and Employment layers, the layered method toString (Address layer) is invoked after the layered method toString (Employment layer) is invoked. This violates the order of message sequence shown in Figure 2. This bug is caused by the usage of the ContextJ® framework consisting of LayerDefinition, define, select, and next. The order of layered method definitions is not correct. Of course, this bug can be easily fixed after the programmer understands the ContextJ® language specifications. However, it is not necessarily easy for a novice to understand the above behavior. If the number of layers and the number of classes associated to the layers increase, it becomes difficult to un-
nderstand the detailed behavior even if the programmer is an expert.

In context-aware systems, it is difficult to check the design consistency, the correspondence between design and its implementation, and non-functional properties specified in a design. That is, it is not easy to check whether a design model is correctly implemented in ContextJ*. Although the structural aspects modeled by class diagrams can be easily mapped to a ContextJ* program, it is hard to check the correspondence between context-dependent behavior modeled by sequential diagrams and actual ContextJ* implementation.

To deal with this problem, we are developing RV4COP, a runtime verification mechanism based on UML4COP. In RV4COP, both a system design model and actual execution trace data at a certain period of time are translated into a logical formula. The validity of a design model, the correspondence between the design and the execution, and the non-functional properties can be verified automatically. For this checking, we use an SMT (Satisfiability Modulo Theories) solver [4], a tool for deciding the satisfiability of logical formulas. SMT generalizes SAT (Satisfiability) by adding equality reasoning, arithmetic, and other first-order theories. Preliminary research results are shown in [11]. Actual execution trace data are collected using AspectJ. In general, it is not easy to apply a formal verification method to trace analysis, because logged data tend to be huge. Our approach, in which only the COP-specific events such as layer in and layer out are collected, can reduce the size of trace data.

6. Conclusion
This paper proposes UML4COP, a UML-based design method for COP. UML4COP and COP improve the expressiveness for designing and implementing context-aware systems. As discussed in this paper, we plan to develop RV4COP towards the next research step.

Acknowledgement
This research is being conducted as a part of the Grant-in-aid for Scientific Research (B), 23300010 by the Ministry of Education, Culture, Sports, Science and Technology, Japan.

References