Mapping Context-Dependent Requirements to Event-Based Context-Oriented Programs for Modularity

Tetsuo Kamina
University of Tokyo
kamina@acm.org

Tomoyuki Aotani
Tokyo Institute of Technology
aotani@is.titech.ac.jp

Hidehiko Masuhara
Tokyo Institute of Technology
masuhara@acm.org

ABSTRACT

There are a number of ways to implement context-dependent behavior, such as conditional branches using `if` statements, polymorphism (such as the state design pattern), aspects in aspect-oriented programming (AOP), and layers in context-oriented programming (COP). The way to implement context-dependent variations of behavior significantly affects the modularity of the obtained applications.

This paper proposes a model of context-dependent requirements and shows the systematic translation from the model to the implementation in the existing COP language EventCJ. The model represents the following facts: (1) abstract contexts, context-dependent use cases, and groups of related use cases called layers; (2) concrete contexts (detailed specification of contexts), context-related external entities, and their correspondence to the abstract contexts; and (3) events that trigger changes of the contexts and thus switch the variations of behavior. We show that all such facts are injectively translated into the program written in EventCJ.

Keywords

Context-oriented programming, Events, Requirements model, Translation to implementation

1. INTRODUCTION

Context-awareness is one of the major concerns in many application areas. It refers to the capability of a system to behave appropriately with respect to its surrounding contexts. A context implies a specific state of a system and/or an environment that affects the system's behavior. For more precise definition, it is identified by observing behavioral changes in the application. An example of context-aware application is a ubiquitous computing application that behaves differently in relation to situations such as geographical location, indoor or outdoor environment, and weather. In this case, some specific states of situations are contexts. An adaptive user interface can also be considered as context-aware as it provides different GUI components (behavior) relative to the current user's task (contexts).

This paper focuses on a methodology to efficiently specify requirements for context-aware applications and systematically implement them. There are a number of ways to implement context-dependent behavior, such as conditional branches using `if` statements, polymorphism (such as the state design pattern), aspects in aspect-oriented programming (AOP), and contexts and layers in context-oriented programming (COP) [12]. The way to implement context-dependent variations of behavior significantly affects the modularity of the obtained applications. For example, a number of such variations are simultaneously activated/deactivated at runtime due to the change in current situation, and implementing them using `if` statements easily crosscut multiple modules. In that case, it is preferable to identify such situations as contexts in COP languages.

Besides context-dependent behavior, a foreseeable control of change of context-dependent behavior is also important. There are complex relations between contexts (that affect the applications behavior) and variations of behavior, which make the modification in behavioral changes with respect to a change in the specification error prone. Thus, systematic identification of contexts and variations of behavior depending on them is required. Furthermore, selecting modularization mechanism for context changes is also important, because context changes are scattered over the whole execution of the application.

In summary, the challenge is to develop a way to coordinate a number of variations of context-dependent behavior and systematically select linguistic mechanisms to implement them.

In this paper, we propose a model of context-dependent requirements and show the systematic translation from that model to the implementation in the existing COP language EventCJ [15]. This model identifies the following requirements specifications: (1) abstract contexts, context-dependent use cases, and groups of related use cases called layers; (2) concrete contexts (detailed specification of contexts), context-related external entities, and their correspondence to the abstract contexts; and (3) events that trigger changes of the contexts and thus switch the variations of behavior. The obtained requirements are directly translated to the implementation with the assumption that the implementation is performed using EventCJ. We formalize this translation to precisely study how the requirements separated in the specifications are mapped into modules in EventCJ, and show that

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright 20XX ACM X-XXXXX-XX-X/XX/XX ...$15.00.

it is mostly injective and performed systematically. Thus, our approach provides modularity in that requirements are not scattered to several modules in the implementation, and each module is not tangled with several requirements.

We demonstrate the effectiveness of this method by conducting two case studies of different context-aware applications. The first one is a conference guide system, which serves a guide for an academic conference including management of a personal attending scheduling, navigation within the venue and around the conference site, and an SNS function such as a Twitter client. The other is CJEdit, a program editor providing different functionalities relative to cursor position [5]. In these case studies, we successfully represented context-related requirements in our model and directly translated these requirements in the implementations in EventCJ.

The remainder of this paper is organized as follows. Section 2 explains the difficulties in development of context-aware applications by using an example of simple pedestrian navigation system, and limitations of the existing approaches. Section 3 describes our requirements model. Section 4 defines mapping from the facts represented by the model into the program written in EventCJ. Section 5 illustrates case studies. Finally, Section 6 concludes this paper.

2. DIFFICULTIES IN CONTEXT-AWARE APPLICATIONS

We explain the difficulties in development of context-aware applications by using a simple pedestrian navigation system implemented on a mobile terminal, which displays the current position of the user. This system changes its behavior according to the situations. When the user is outdoors, it displays a city map, which is updated whenever the current position of the user is changed. When the user is inside a building where a specific floor plan service is provided, it displays a floor plan of that building. When the user is inside a building where no such services are provided, it displays the city map as in the case where the user is outdoors. If no positioning systems are available, it displays a static map.

Identification of context-dependent behavior.

A context-aware application changes its behavior with respect to current executing context; i.e., there are a number of variations of behavior depending on contexts. Thus, we need to identify contexts and requirements variability depending on them. For example, in the pedestrian navigation system, we can identify contexts such as outdoors or indoors, the availability of the special floor plan services, and the availability of the positioning systems. These contexts change the behavior of the map and other functions such as GUI components. For example, the variation of behavior "displaying a city map" is selected when the user is "outdoors."

Requirements volatility in context sensing.

Technologies for sensing context changes are very complex and continuously evolving. This means that requirements specification for context sensing are subject to change. For example, at first, it seems appropriate to define the outdoors/indoors situations on the basis of status of the GPS receiver. However, this definition may change in the future to use the air pressure sensor or other technologies that are not currently implemented in the smartphone (such as an active RFID receiver).

Different levels of abstraction.

As discussed in the volatility in context sensing, contexts at the abstract level consist of multiple concrete contexts. For example, the availability of positioning systems depends on the hardware specifications such as the availability of GPS and/or wireless LAN functions. Thus, we need to precisely define what are contexts in terms of the target machine. This chain of dependency leads to the difficulty in precise definition on when the variations of behavior switches at runtime. For example, there may be multiple state changes in the target machine that trigger a context change, because some states of executing hardware may barrier or guard the change of abstract contexts.

Crosscutting of contexts in requirements.

In context-aware applications, a number of variations of behavior that are at first irrelevant to each other may be eventually considered relevant in that they are executable in the same context. For example, we may also identify variations of behavior for other functions such as GUI components. The variation "displaying an alert message on the status bar" may be considered relevant to "displaying a static map" if the former is executable only when no positioning systems are currently available.

Multiple dependency between contexts and behavior.

We also need to carefully analyze dependency between contexts and variations of behavior, because a number of variations depend on multiple contexts. For example, according to the problem description, the variation "displaying a city map" depends both on outdoors/indoors situations and the availability of the special floor plan service. In this case, this variation is not selected if either or both situations "the user's situation is outdoors" and "the floor plan service is not available" are not satisfied. In general, multiple contexts may barrier or guard the execution of context-dependent behavior. This dependency becomes more complicated when we consider the concrete contexts as discussed above.

Crosscutting of behavioral changes in requirements.

One of the most important properties of context-aware applications is that they change their behavior at runtime. Thus, we need to identify when a variation of behavior switches to another one. As discussed above, however, a variation may depend on multiple (abstract) contexts, where each context depends on a number of concrete contexts. In particular, context changes are scattered over multiple requirements. Since their specifications are subject to change, it is desirable to localize them.

Translation to modular implementation.

The above difficulties (from the viewpoint of requirements) make it difficult to separately translate requirements to the implementation. We need to carefully trace which requirements are implemented by which modules. It is also desirable if a module in the implementation is not tangled with multiple requirements but it implements only a single requirement. Thus, to support modularity, it is desirable that

there is an injective mapping from requirements to the implementation.

2.1 Existing Approaches

COP languages provide a novel linguistic construct to modularize context-dependent behavior called layers. A number of COP languages have been developed thus far, and some of them share the same abstraction mechanism based on layers and partial methods [4, 6, 9, 11]. On the other hand, Few research efforts are devoted to systematize the design of context-oriented programs. For example, it is not unclear how to discover layers from requirements, and when using layers is more preferable than using existing object-oriented mechanisms and `if` statements to implement context-dependent behavior. For dynamic activation of layers, a number of mechanisms have been proposed in COP. Most of the existing COP languages are based on a dynamically scoped layer activation mechanism using so-called `with`-blocks, which makes context activation code scattered over the whole program. Event-based activation of layers with the support of aspect-oriented programming (AOP) features are proposed to separate the control of layer activation from the base program [15, 6]. All these mechanisms are useful with the assumption that we already determine what are contexts and what are behavioral variations depending on them. We require a software development methodologies that addresses aforementioned difficulties.

There have been a number of software development methodologies. Object-oriented methodologies are useful to discover objects and classes from the requirements and analyze them. Aspect-oriented software development (AOSD) methodologies [14, 21] are useful to find crosscutting concerns and modularize them. Feature-oriented software development (FOSD) [3] is a method that maps feature diagrams [17], which are obtained from the analysis of software to be developed, to implementations. Feature diagrams are useful for analyzing dependency among features from which software is constructed. Even though these methodologies provide a good starting point to consider how we develop context-aware applications, they do not focus on the solutions for the aforementioned difficulties. We need to extend the existing methodologies to systematically identify contexts and behavior depending on them to provide a foreseeable control of change of context-dependent behavior.

Recently, a number of approaches to discover, analyze, and implement contexts and variations of behavior depending on them have been studied. Several requirements engineering methods [24, 23, 25, 19, 20, 2] mainly focus on discovery and analysis of (abstract) contexts and variations of behavior depending on them. Henrichsen and Indulska proposed a software engineering framework for pervasive computing [10]. They do not provide any systematic ways to manage volatile requirements for concrete context, and to modularly implement them. Specifically, they do not identify a set of variations that comprises one single module. Frameworks and libraries for context-aware applications provide context-aware software components and thus enhances reusability, addressing some of the difficulties mentioned above [8, 7, 1, 22]. They are domain-specific and any general solutions for context-aware applications are provided.

3. A MODEL OF CONTEXT-DEPENDENT REQUIREMENTS

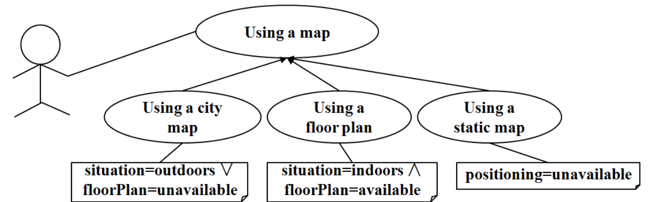

Figure 1: Use case diagram for the pedestrian navigation system

We propose a model of context-dependent requirements that represents the following requirements specifications:

1. Abstract contexts, use cases that depend on them, and groups of related use cases called layers
2. Concrete contexts (detailed specification of contexts), context-related external entities, and their correspondence to the abstract contexts
3. Events that trigger changes of the contexts and thus switch the variations of behavior

3.1 Abstract contexts

A context in our model is defined in terms of variables that take finite states (values). By observing the behavior (such as use cases) of the pedestrian navigation system, we identify the following variables:

name	values
situation	*outdoors*, *indoors*
floorPlan	*available*, *unavailable*
positioning	*available*, *unavailable*

Each of those variables corresponds to the situation for using the system, the availability of the floor plan service, and the availability of the positioning devices, respectively. In the following sections, we call a specific setting of value to a variable (i.e., a state of the variable) as a context.

3.2 Context-dependent use cases

Our model identifies context-dependent use cases. A context-dependent use case is a use case annotated with a proposition that specifies when it is executable. In general, a context-dependent use case specializes another use case. For example, in the pedestrian navigation system, we can identify a use case "using a map" (Map). We can then identify three context-dependent use cases "using a city map" (CityMap), "using a floor plan" (FloorPlan), and "using a static map" (StaticMap). All these context-dependent use cases are specialization of Map. CityMap is annotated with the proposition *situation*=*outdoors*∨*floorPlan*=*unavailable*, which means that it is executable only when the value of *situation* is *outdoors* or the value of *floorPlan* is *unavailable*. Similarly, FloorPlan is annotated with *situation*=*indoors*∧ *floorPlan*=*available*, and StaticMap is annotated with *positioning*=*unavailable* (Figure 1).

3.3 Grouping context-dependent use cases

Context-dependent use cases that are executable under the same context are grouped into one *layer*. In general, a system consists of a number of use cases. Figure 2 shows a

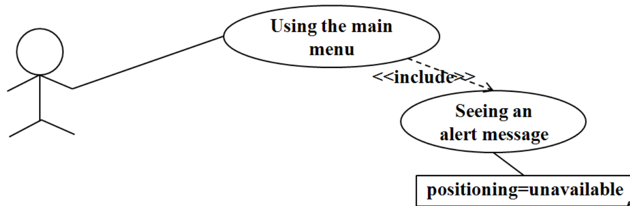

Figure 2: The context-dependent use case that displays an alert message

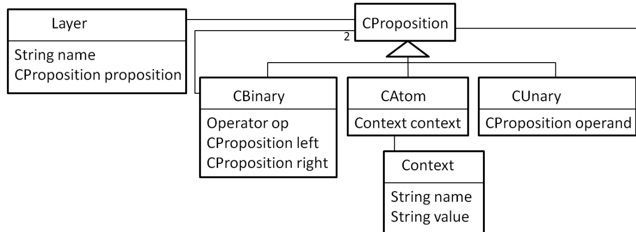

Figure 3: The model of layers

use case diagram for another interaction between the pedestrian navigation system and the user, namely "using the main menu." There is a context-dependent use case, namely "seeing an alert message" (Alert), which describes the behavior of the pedestrian navigation system that displays an alert message indicating that no positioning systems are available on the status bar. Note that this use case is annotated with the same condition as StaticMap. This means that the use cases StaticMap and Alert are executable under the same context. To provide better maintainability, our model groups such use cases into one layer[1].

We show the model of layers by using the UML class diagram in Figure 3. A layer consists of its name and the proposition annotated to the constituent context-dependent use cases (in this diagram, we omit context-dependent use cases, because they share the same proposition within the same layer). This proposition is represented by the class `CProposition`, which has three subclasses. `CBinary` represents binary operators $\wedge$ and $\vee$, `CUnary` represents the unary operator $\neg$, and `CAtom` represents a ground term, which is the name of context and its value represented by the class `Context`. Note that we consider the contexts that share the same name but have different values as different instances; i.e., each field of `Context` is considered "final."

3.4 Specifying concrete contexts

While contexts are abstract from the viewpoint of behavioral variations, when we identify requirements from the viewpoint of context sensing, we need to consider more concrete level of contexts. We firstly list all resources of the running machine and external entities that are relevant to the context-dependent behavior. For example, we list the following resources and external entities for the pedestrian navigation system:

[1]Each layer directly corresponds to a `layer` declaration in existing COP languages.

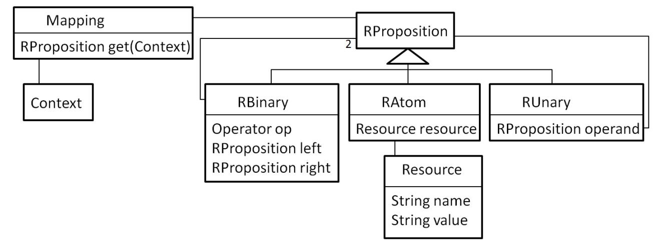

Figure 4: The model of resources and their mapping

- Resources: GPS, Wi-Fi
- External entities: the floor plan services (FP)

We refer them as *concrete contexts*. Then, we map the pairs of concrete contexts and their values to those of abstract contexts and their values. By analyzing when the system is in each context with respect to the status of concrete contexts, we can create a mapping from abstract contexts to concrete contexts. For example, the value of *positioning* is *available* when the GPS device is switched on, or the Wi-Fi device is connected to the Internet. Table 1 summarizes this mapping.

The model of concrete contexts and their mapping to abstract contexts is shown in Figure 4. Each concrete context (represented as `Resource` in Figure 4) consists of its name and value. As in the case of abstract contexts, we distinguish a concrete context that has the same name but provides the different value as a different instance. The class `Mapping` provides the function `get` that takes an instance of `Context` and returns a proposition, which is represented by the class `RProposition`. This class has three subclasses `RBinary`, `RUnary`, and `RAtom` to represent binary operators, the unary operator, and ground terms, respectively.

3.5 Identifying Events

In terms of concrete level of contexts, we identify events that change those contexts. Each event consists of its name, a pair of a concrete context and its value before the state change occurs, a pair of the concrete context and its value after the state change occurs, and the description about when this state change occurs. For example, the value of the GPS becomes "over the criterion value" from "under the criterion value" when the received GPS signal value becomes greater than the preset value; we can identify this state change as

Table 1: Mapping from contexts to machine-level resources in the pedestrian navigation system

context	value	resource configuration
situation	*outdoors*	GPS=over the criterion
	indoors	GPS=under the criterion
floorPlan	*available*	FP=exists
	unavailable	FP=do not exists
positioning	*available*	GPS=on or Wi-Fi=connected
	unavailable	GPS=off and Wi-Fi=disconnected

Table 2: Examples of the identified events

name	transition	when
StrongGPS	GPS=over the criterion → GPS=under the criterion	the GPS signal value becomes under XXX
GPSEvent	GPS=off → GPS=on	the GPS device is becoming on
WifiEvent	Wi-Fi=disconnected → Wi-Fi=connected	the Wi-Fi device is connected and the IP address is properly set

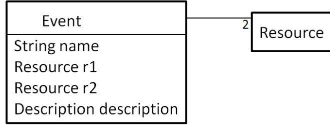

Figure 5: The model of events

an event with the name StrongGPS. We list some examples of the events identified in the pedestrian navigation system in Table 2.

The model of events is shown in Figure 5. Each event consists of its name, concrete contexts before and after the event is generated, and its description.

4. TRANSLATING TO THE IMPLEMENTATION

This section discusses how the specifications represented by our model are separately translated to each linguistic construct of the existing COP language, namely EventCJ [15, 16]. To make this paper self-contained, we firstly provide a short introduction to EventCJ. Then, we define the translation from our model to EventCJ to demonstrate how this translation is systematically performed.

4.1 Short Introduction to EventCJ

As in other COP languages, layers and partial methods comprise the mechanism for modularization of context-dependent behaviors in EventCJ.

Figure 6 shows an example of layers and partial methods in EventCJ that are responsible for displaying a map in the pedestrian navigation system. The class `Navigation` declares the method `run` that updates the map. `Navigation` also declares three layers, namely `CityMap`, `FloorPlan`, and `StaticMap`. `CityMap` defines the behavior of the map when the system is outdoors; `FloorPlan` defines the behavior of the map when there is a special floor plan service; and `StaticMap` defines the behavior when there are no available positioning devices. All layers extend the original behavior of `run` by declaring *around* partial methods, which are executed instead of the original `run` method when the respective layer is active[2].

In COP languages, we can dynamically activate and deactivate layers. For this purpose, EventCJ provides the `when`

[2]There are also *before* and *after* partial methods that execute before and after the execution of the original method, respectively, when the respective layer is active.

```
1  class Navigation extends MapActivity
2      implements Runnable, LocationListener {
3    MyLocationOverlay overlay;
4    void onStatusChanged(..) { .. }
5    void run() {}
6    void onCreate(Bundle status) {
7      .. overlay.runOnFirstFix(this); ..
8    }

10   layer CityMap when StrongGPS || !FPExists {
11     void run() { .. }
12   }
13   layer FloorPlan
14     when !StrongGPS && WifiConnect && FPExists {
15     void run() { .. }
16   }
17   layer StaticMap when !GPSon && !WifiConnect {
18     void run() { .. }
19   }
20 }
```

Figure 6: Layers and partial methods in EventCJ

clauses in the layer declarations (this feature is available from the later version of EventCJ [16]), layer transition rules, and events.

The `when` clauses control the implicit activation of layers. If a layer is declared with a `when` clause (as shown in Figure 6), it implicitly becomes active when the proposition specified by the `when` clause becomes true. In this proposition, each ground term is the name of a layer (true when active). For example, the layer `CityMap` is active only when `StrongGPS` is active or `FPExists` is not active. We can use the logical operators `||`, `&&` and `!` to compose propositions.

A layer that does not have a `when` clause is called a *context* (we may declare partial methods and other members in such a layer. In this example, though, all such layers have an empty body):

```
layer StrongGPS {}
layer WifiConnect {}
layer FPExists {}
layer GPSon {}
```

The activation of such layers are controlled by *layer transition rules*, which are triggered by events (explained below). Examples of layer transition rules upon events `GPSEvent` and `WifiEvent` are as follows:

```
transition GPSEvent: -> GPSon
transition WifiEvent: -> WifiConnect
...
```

Each rule starts from the keyword `transition`, and is followed by an event name and a rule. The left-hand side of the `->` operator (omitted in this example) consists of contexts to be deactivated, and the right-hand side consists of contexts to be activated. We may add a guard for the rule by putting the `?` operator at the left hand side of `->`, which is also omitted in this example. We may concatenate multiple subrules by the `|` operator; in such a case, only the leftmost applicable rule is applied. Thus, the first rule above is read as, "upon the generation of `GPSEvent`, the layer `GPSon` is activated."

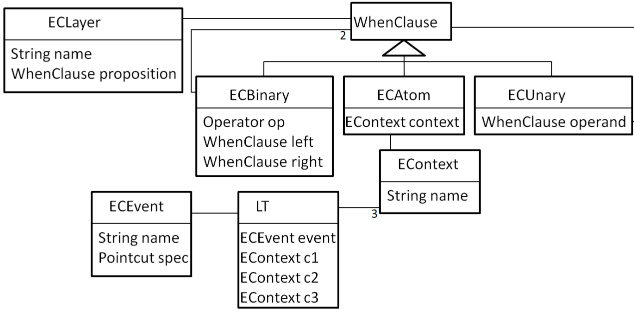

Figure 7: The model of EventCJ

EventCJ provides events to trigger the layer transition rules. The following code fragment shows a declaration of event `GPSEvent`:

```
declare event GPSEvent(Navigation n, int s)
  :after call(void Navigation.onStatusChanged(s))
    &&target(n)&&args(s)
    &&if(s==LocationProvider.AVAILABLE);
```

An event declaration consists of two parts: a specification that indicates when the event is generated and a specification that indicates where the event is sent. The former is specified by using AspectJ-like pointcut sublanguage [18], and the latter is specified by using the `sendTo` clause that lists instances that receive the event. For example, `GPSEvent` specifies when it is generated by using the pointcut specification that specifies a join-point just after the `onStatusChanged` method on `Navigation` is called with the argument value indicating that the location provider is available. In this example, the `sendTo` clause is omitted, which means that the effect of the event is *global*, i.e., it changes the behavior of all classes in the program.

To discuss the correspondence between our model and EventCJ, we show the metamodel of EventCJ programs in Figure 7. Each layer, represented by the class `ECLayer`, consists of its name and the `when` clause, which is represented by the class `WhenClause` (we abstract other irrelevant constructs such as classes from this metamodel). This class has three subclasses: `ECBinary`, `ECUnary`, and `ECAtom`. `ECAtom` represents a ground term for the `when` clauses, which is a context (i.e., a layer that does not have a `when` clause). This is represented by the class `ECContext`. `ECBinary` and `ECUnary` represent binary operators and the unary operator for the `when` clauses, respectively. An event, represented by `ECEvent`, consists of its name and the specification about when this event is generated written in the pointcut language. A layer transition rule, represented by `LT`, consists of the corresponding event, a context that guards the transition (c1), a context that is deactivated (c2), and a context that is activated (c3).

4.2 Definition of mapping

To precisely study how the requirements separated in the specifications are translated into modules in EventCJ, and which parts of this translation can be mechanized, we formally define the translation from our model to EventCJ. First, concrete contexts represented in our model are mapped to contexts in EventCJ. We define a function map_R that maps each instance of `Resource` in Figure 4 to an instance of `ECContext` in Figure 7. This function is injective. Note that this function may be a partial function, because some concrete contexts take just two exclusive values and thus we need to identify only one context in EventCJ. Example mappings defined for the pedestrian navigation system are as follows (we represent an instance of `ECContext` by its name in the typewriter format):

$$\begin{aligned} map_R(\text{GPS=over the criterion value}) &= \texttt{StrongGPS} \\ map_R(\text{GPS=on}) &= \texttt{GPSon} \\ map_R(\text{Wi-Fi=connected}) &= \texttt{WifiConnect} \\ map_R(\text{FP=exists}) &= \texttt{FPExists} \end{aligned}$$

This mapping should be manually defined by the developer.

Next, we define the mapping from layers in the model to those in EventCJ. For this purpose, we define the function map_L that takes an instance of `Layer` in Figure 3 and returns an instance of `ECLayer` in Figure 7. The returned instance consists of a name mapped from the name of the argument instance, and the `when` clause that is mapped from the corresponding context annotation `CProposition` in Figure 3 (let l be an instance of `Layer`):

$$\begin{aligned} map_L(l) &= \texttt{new ECLayer}(id(l.name), \\ &\quad map_C(l.annotation)) \end{aligned}$$

For the name mapping, we assume the identity function id that takes a string text and returns it. We further need to elaborate how to map an instance of `CProposition` to that of `WhenClause`, which is defined by the function map_C. Since `CProposition` is an abstract class, we need to define the cases for each concrete class. If the instance of `CProposition` is a composite proposition, i.e., that is an instance of either `CUnary` or `CBinary`, we define the map function as follows (let c, c_1, and c_2 be instances of `CProposition`):

$$\begin{aligned} map_C(c_1 \wedge c_2) &= map_C(c_1) \wedge map_C(c_2) \\ map_C(c_1 \vee c_2) &= map_C(c_1) \vee map_C(c_2) \\ map_C(\neg c) &= \neg map_C(c) \end{aligned}$$

If the instance of `CProposition` is an atom, we obtain the corresponding proposition by the class `Mapping` in Figure 4, and map it to the `when` clause:

$$map_C(c) = map_{RP}(\texttt{Mapping}.get(c.context))$$

The *get* function returns a proposition (an instance of `RProposition` in Figure 4), which is mapped to an instance of `WhenClause` in Figure 7 by the map_{RP} function at the right-hand side. Since `RProposition` is an abstract class, we also need to define the cases for each concrete class. We only show the case when the instance of `RProposition` is `RAtom` (let r be an instance of `RProposition`):

$$map_{RP}(r) = \texttt{new ECAtom}(map_R(r.resource))$$

It firstly maps a resource to a context in EventCJ, and creates an instance of `ECAtom`.

The mapping from events (in the model) to events (in EventCJ) is obvious (let e be an instance of `Event`):

$$\begin{aligned} map_E(e) &= \texttt{new ECEvent}(\\ &\quad id(e.name), \\ &\quad map_R(e.r1), map_R(e.r2), map_D(e.d)) \end{aligned}$$

It maps the name, resources, and the description to the corresponding constructs in EventCJ, and creates an instance

of `ECEvent`. For the name mapping, we may assume the identity function. The description is mapped to the corresponding pointcut expression. This mapping is manually performed by the developer. We may apply the method to identify AspectJ's pointcut from the *extension pointcut* in use cases, described in [14].

The events in the model are also mapped to layer transition rules. For this purpose, we define the function map_{LT} that takes an instance of `Event` and returns an instance of `LT` in Figure 7:

$$map_{LT}(e) = \texttt{new LT(} \texttt{new ECEvent}(id(e.name), map_D(e.d)), map_R(e.r1), map_R(e.r1), map_R(e.r2))$$

For the concrete implementation, we need to populate definitions of classes, methods, and partial methods into the source code. Designing base code (i.e., classes and methods) from use cases is fully discussed in [13], and we do not describe it in detail in this paper. The method for designing layers is a straightforward extension of [13].

This mapping is mostly mechanized. The developer needs to provide the name mapping from resources to contexts (in EventCJ) and pointcut expressions for each event. However, we may automate other parts. Furthermore, all the map_X functions are injective. Thus, this mapping provides modularity in that requirements are not scattered to several modules in the implementation, and each module is not tangled with several requirements.

5. CASE STUDIES

This section demonstrates the applicability of our model to development of context-aware applications through two case studies.

5.1 Conference Guide System

The first case study is a conference guide system, which serves a guide for an academic conference. This system is implemented on an Android smartphone, and provides the conference program, management of personal attending scheduling, navigation inside the venue and around the conference site, an alarm function that notifies the user if the user is out of the session room when the starting time of the session is approaching, and a Twitter client to enable the user to have a live of the conference. This system has a couple of context-related behavioral variations listed as follows:

- The conference program is provided online; the user can view the online program when the Internet is available for the smartphone. The downloaded program is cached on the local database if no cached program is available. The user can view the cached program when no Internet connections are available. The user cannot view the program if there is no cached program and no Internet connections are available.
- From the program, the user can select sessions that she will attend. The selected sessions are listed on the personal attending schedule. The listing of the selected sessions is available only when there are some selected ones.
- The system provides a map function. When the user is within the conference venue, the map provides a floor plan of that venue. When the user is out of the venue, it provides a city map around the conference site, which is updated when the new position of the user is detected. The positioning is performed based on GPS or the Wi-Fi connection. If no positioning devices are available, it provides a static map around the conference site.
- The system provides a Twitter client, which is available only when the Internet is available.

According to these problem descriptions, we identify the following contexts:

name	values
Internet	*available*, *unavailable*
cache	*available*, *unavailable*
schedule	*available*, *unavailable*
situation	*outdoors*, *indoors*
positioning system	*available*, *unavailable*

Then, we identify a number of context-dependent use cases. For example, we identify following context-dependent use cases by considering the use case for viewing conference program (each proposition in parenthesis is the proposition for the use case):

- "viewing online program" (*Internet*=*available*)
- "viewing offline program" (*Internet*=*unavailable*$\wedge$ *cache*=*available*)

Another example of context-dependent use case is "using a Twitter client," which is applicable only when the Internet is available:

- "usint a Twitter client" (*Internet*=*available*)

Some of the context-dependent use cases share the same condition. For example, both use cases "viewing online program" and "using a Twitter client" are executable only when the condition *Internet*=*available* is true. We group these context-dependent use cases into a layer `WithInternet`.

Then, we construct a mapping from contexts to resources. For example, we show the mapping from *Internet* and corresponding resources. Assuming that the target smartphone is equipped with Wi-Fi and mobile network devices, the mapping is defined as shown in the following table.

context	value	resource configuration
Internet	available	Wi-Fi=connected or Mobile=connected
	unavailable	Wi-Fi=disconnected and Mobile=disconnected

We also define the mapping for other contexts in a similar manner. Then, events that change contexts are identified by listing the state changes of the resources.

All these facts discovered above are mapped onto the implementation in EventCJ. We firstly identify the contexts in EventCJ from the resources. Followings are examples of contexts that deal with the Internet connectivity:

```
layer Wifi {}  /* Wi-Fi=connected */
layer Mobile{} /* Mobile=connected */
```

Since those resources take just two exclusive values, "connected" or "disconnected," we only define one context for each resource. The other case is represented by the negation of the defined context.

Layers identified in our model are mechanically translated to layer declarations in EventCJ. The following layer declaration is obtained by applying the mapping method explained in Section 4.2 to the layer `WithInternet` discovered above:

```
layer WithInternet when Wifi || Mobile {
  Cursor getProgram() { .. }
}
```

The `when` clause is obtained by translating the proposition for the context-dependent use case. The body of layer declaration is populated by some context-dependent behavior. For example, how to obtain the conference program (implemented by the method `getProgram`) changes with respect to the current execution context. Thus, we implement this method by using a set of partial methods.

Events are translated into event declarations and layer transition rules in EventCJ. We show an implementation of the event that switches the value of "Wifi" to be "connected" as an example:

```
event WifiEvent(ConferenceGuide cg)
  :after call(void ConferenceGuide.onResume())&&
    if(cg.getMng().getNetworkInfo().getState()
       == NetworkInfo.DetailedState.CONNECTED);
```

This event is generated when the `onResume` method declared in the `Activity` class (provided by the Android SDK framework), which is the superclass of `ConferenceGuide`. It checks the current status of network connection, and if it is configured with an appropriate IP address, a `WifiEvent` is generated. The corresponding layer transition rule is as follows, which activates `Wifi` when this event is generated:

```
event WifiEvent: -> Wifi
```

5.2 CJEdit

CJEdit is a program editor that enhances the readability of programs by providing different text formatting techniques for code and comments. The code part is formatted in a typewriter format with syntax highlighting, and the comment part is formatted in a rich text format (RTF) that supports multiple fonts, text sizes, decorations, and alignments. Furthermore, CJEdit provides different GUI components depending on whether the programmer writes code or comments. This application is firstly implemented by Appeltauer [5]. We take this example to investigate how our model fits the development of the existing context-aware application.

Since there already exists the original implementation of CJEdit, we do not perform this case study from scratch. We use the original implementation as a prototype of this case study, and by observing the system's behavior, we firstly derive contexts listed as follows:

name	values
cursor	*onCode*, *onComments*, *none*
textRegion	*code*, *comments*

The *cursor* takes three states: the cursor is on code (*onCode*), the cursor is on comments (*onComments*), and the special case that the cursor does not exist on the editor (e.g., before clicking the editor after starting the application). The text region (textRegion) takes two states: *code* and *comments*.

In CJEdit, we identify the use case "editing a program," which includes another use case "displaying the source code." We derive context-dependent use cases from these use cases. "Editing a program" executes different use cases with respect to the cursor's position; "writing code" is executable only when the condition *cursor=onCode* is true, and "writing comments" is executable only when the condition *cursor=onComments* is true. "Displaying the source code" executes three different use cases with respect to the text region and the cursor's position; "with syntax highlighting" is executable only when the condition *cursor=onCode*$\wedge$ *textRegion=code* is true; "without syntax highlighting" is executable only when the condition *cursor=onComments*$\wedge$ *textRegion=code* is true; "RTF format" is executable only when the condition *textRegion=comments* is true. Since all the conditions for context-dependent use cases are distinct, we identify a layer for each context-dependent use case.

The next step is to identify resources and external entities that affects the values of the contexts, and to construct a mapping from contexts to those resources and external entities. In the case of CJEdit, however, we found that there is nothing to do at this step, because this application does not depend on any specific low-level devices of the target machine. Thus, we define events by listing the state changes of the aforementioned contexts.

We show that all the facts obtained above are mapped into the implementation in EventCJ. First, we identify the following contexts:

```
layer OnCode {}      /* cursor=onCode */
layer OnComments{}  /* cursor=onComments */
layer RndCode {}     /* textRegion=code */
```

Since the cursor takes three states, we identify two of them as contexts. For the text region, we identify just one context, since it takes two alternative states.

Layers identified in our model are mechanically translated to layer declarations. For example, layers implementing context-dependent use cases "writing code" and "writing comments" are implemented by layers `CodeEditing` and `CommentEditing`:

```
layer CodeEditing when OnCode {
  after void showWidgets() { .. }
  after void showToolbars() { .. }
}
layer CommentEditing when OnComments {
  after void showMenu() { .. }
  after void showToolbars() { .. }
}
```

In each layer, a `when` clause is translated from the annotations in the corresponding context-dependent use case. The body of each layer is populated with partial methods. They provide partial methods that modify the behaviors of the original methods for displaying widgets (`showWidgets`), toolbars (`showToolbars`), and menus (`showMenu`).

Events are translated into event declarations and layer transition rules in EventCJ. We show an implementation of the event that switches the value of the context "cursor" to be "onCode" as an example:

```
event MoveOnCode(TextEditor edit)
  :after execution(
      void TextEditor.onCursorPositionChanged())
    && this(edit);
```

The layer transition rule triggered by this event is defined as follows:

```
transition MoveOnCode:
  OnComments ? OnComments -> OnCode
| -> OnCode;
```

We implement other events and layer transition rules in a similar manner.

6. CONCLUDING REMARKS

In this paper, we proposed a model of context-dependent requirements. The model is well expressive to describe variations of behavior with respect to the abstract contexts, as well as to represent more concrete and volatile requirements about context sensing. This model supports modularity in that the specifications represented by the model are injectively translated to the program written in the existing event-based context-oriented language.

This paper only describes the model of requirements and its mapping to the implementation by using simple examples. How to document the requirements based on this model in more sophisticated case studies remains as future work.

7. REFERENCES

[1] Gregory D. Abowd, Christopher G. Atkeson, Jason Hong, Sue Long, Rob Kooper, and Mike Pinkerton. Cyberguide: A mobile context-aware tour guide. *Wireless Networks*, 3(5):421–433, 1997.

[2] Raian Ali, Fabiano Dalpiaz, and Paolo Giorgini. Goal-based self-contextualization. In *CAiSE 2009*, pages 37–43, 2009.

[3] Sven Apel and Christian Kästner. On overview of feature-oriented software development. *Journal of Object Technology*, 8(5):49–84, 2009.

[4] Malte Appeltauer, Robert Hirschfeld, Michael Haupt, and Hidehiko Masuhara. ContextJ: Context-oriented programming with Java. *Computer Software*, 28(1):272–292, 2011.

[5] Malte Appeltauer, Robert Hirschfeld, and Hidehiko Masuhara. Improving the development of context-dependent Java application with ContextJ. In *COP'09*, 2009.

[6] Malte Appeltauer, Robert Hirschfeld, Hidehiko Masuhara, Michael Haupt, and Kazunori Kawauchi. Event-specific software composition in context-oriented programming. In *Proceedings of the International Conference on Software Composition 2010 (SC'10)*, volume 6144 of *LNCS*, pages 50–65, 2010.

[7] Cinzia Cappiello, Marco Comuzzi, Enrico Mussi, and Barbara Pernici. Context-management for adaptive information systems. *Electronic Notes in Theoretical Computer Science*, 146:69–84, 2006.

[8] Stefano Ceri, Florian Daniel, Federico M. Facca, and Maristella Matera. Model-driven engineering of active contet-awareness. *World Wide Web*, 10:387–413, 2007.

[9] Pascal Costanza and Robert Hirschfeld. Language constructs for context-oriented programming – an overview of ContextL. In *Dynamic Language Symposium (DLS) '05*, pages 1–10, 2005.

[10] Karen Henrichsen and Jadwiga Indulska. A software engineering framework for context-aware pervasive computing. In *PERCOM'04*, 2004.

[11] Robert Hirschfeld, Pascal Costanza, and Michael Haupt. An introduction to context-oriented programming with ContextS. In *GTTSE 2007*, volume 5235 of *LNCS*, pages 396–407, 2008.

[12] Robert Hirschfeld, Pascal Costanza, and Oscar Nierstrasz. Context-oriented programming. *Journal of Object Technology*, 7(3):125–151, 2008.

[13] Ivar Jacobson, Magnus Christerson, Patrik Jonsson, and Gunnar Övergaard. *Object-Oriented Software Engineering: A Use Case Driven Approach*. Pearson Education, 1992.

[14] Ivar Jacobson and Pan wei Ng. *Aspect-Oriented Software Development with Use Cases*. Pearson Education, 2005.

[15] Tetsuo Kamina, Tomoyuki Aotani, and Hidehiko Masuhara. EventCJ: a context-oriented programming language with declarative event-based context transition. In *AOSD '11*, pages 253–264, 2011.

[16] Tetsuo Kamina, Tomoyuki Aotani, and Hidehiko Masuhara. Introducing composite layers in EventCJ. *IPSJ Transactions on Programming*, 6(1):1–8, 2013.

[17] Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, 1990.

[18] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Grisword. An overview of AspectJ. In *ECOOP'01*, pages 327–353, 2001.

[19] Alexiei Lapouchnian and John Mylopoulous. Modeling domain variability in requirements engineering with contexts. In *ER 2009*, volume 5829 of *LNCS*, pages 115–130, 2009.

[20] Sotirious Liaskos, Alexei Lapouchnian, Yijun Yu, Eric Yu, and John Mylopoulos. On goal-based variability acquisition and analysis. In *RE'06*, pages 79–88, 2006.

[21] Awais Rashid, Peter Sawyer, Ana Moreira, and João Araújo. Early aspects: a model for aspect-oriented requirements engineering. In *RE'02*, pages 199–202, 2002.

[22] Daniel Saliber, Anind K. Dey, and Gregory D. Abowd. The context toolkit: Aiding the development of context-enabled applications. In *CHI'99*, pages 434–441, 1999.

[23] Mohammed Salifu, Bashar Nuseibeh, Lucia Rapanotti, and Thein Than Tun. Using problem descriptions to represent variability for context-aware applications. In *VaMoS 2007*, 2007.

[24] Mohammed Salifu, Yujun Yu, and Bashar Nuseibeh. Specifying monitoring and switching problems in context. In *RE'07*, pages 211–220, 2007.

[25] Alistair Sutcliffe, Stephen Fickas, and McKay Moore Sohlberg. PC-RE: a method for personal and contextual requirements engineering with some experience. *Requirements Engineering*, 11(3):157–173, 2006.