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 several challenges in development of context-aware systems. First, while contexts are abstract from the viewpoint of behavior that depends on them, we need to elicit more concrete level of contexts that are sensed by complex context sensing technologies. Second, there are complicated relations between contexts and behavioral variations. Several variations may depend on multiple contexts, and several behavioral variations crosscut across several requirements. Third, contexts and context-dependent behavior reactively change with respect to external/internal events, and these changes also crosscut across several requirements. Finally, these complexities in requirements make it difficult to modularly map requirements to the implementation. This paper proposes a model of context-dependent requirements and shows the modular mapping 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 level 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 issues in many application areas including reactive systems. It refers to the capability of a system to behave appropriately with respect to its surrounding contexts. A context is a specific state of a system and/or an environment that affects the system's behavior. For example, a ubiquitous computing application behaves differently in relation to contexts such as geographical location, indoor or outdoor environment, and weather. An adaptive user interface can also be considered as context-aware as it provides different GUI components relative to the current users task.

There are several difficulties in the development of context-aware applications. First, while contexts are abstract from the viewpoint of behavior that depends on them, we need to elicit more concrete level of contexts that depend on the underlying sensor technologies. Thus, we need to manage different levels of abstraction of contexts in the requirements. Second, there are complicated relations between contexts and behavioral variations. Several variations may depend on multiple contexts, and several behavioral variations crosscut across several requirements. Third, contexts and context-dependent behavior reactively change with respect to external/internal events, and these changes also crosscut across several requirements. In particular, technologies for sensing external events that change contexts evolve continuously, which means that context sensing is a subject to change and thus it should be modularized. Finally, these complexities in requirements make it difficult to modularly map requirements to the implementation.

There are several research efforts that address these difficulties; they are devoted in requirements engineering [27, 26, 29, 21, 22, 2], frameworks [1, 25], and programming languages such as context-oriented programming (COP) languages [14, 4, 5, 9, 13, 28, 17] and event-based languages [24, 10, 23, 11]. These research efforts address only some parts of the aforementioned difficulties. For example, most of the requirements engineering methods for context-aware systems lack the viewpoint of fine-grained and volatile requirements about context sensing, and that of how requirements are modularly mapped to the implementation. Similarly, the implementation technologies lack the viewpoint of requirements.

In this paper, we propose a model of context-dependent requirements and show the modular mapping from the model to the implementation in the existing COP language EventCJ [17]. This model identifies the following facts: (1) abstract contexts, context-dependent use cases, and groups of related use cases called layers; (2) concrete level 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 modularly mapped to the implementation with the assumption that the implementation is performed by EventCJ. By formally define this mapping, we show that 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.

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. 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 discusses related work. 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, or, 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 several 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.

In context-aware applications, contexts change dynamically. While contexts are abstract from the viewpoint of variability of behavior, technologies for sensing context changes are very complex. These technologies evolve continuously, which means that requirements for context sensing are subject to change.

Different levels of abstraction.

As discussed in the volatility in context sensing, contexts at the abstract level consist of several 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 several state changes in the target machine that trigger a context change, because, as in the case of contexts, some states of executing hardware may barrier or guard the change of abstract contexts.

Crosscutting of contexts in requirements.

In context-aware applications, several 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 several 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 one of the contexts "the user's situation is outdoors" and "the floor plan service is not available" is not satisfied. In general, several 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 several concrete contexts. In particular, context changes are scattered in several requirements. Since their specifications are subject to change, it is desirable to localize them.

Modular translation to the implementation.

The above difficulties (from the viewpoint of requirements) make it difficult to modularly map 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 several requirements but it serves only a single requirement. Thus, to support modularity, it is desirable that there is an injective mapping from requirements to the implementation.

3. A MODEL OF CONTEXT-DEPENDENT REQUIREMENTS

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

1. Abstract contexts, use cases that depend on them, and groups of related use cases called layers

2. Concrete level 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

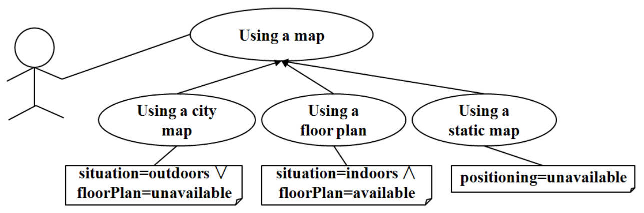

Figure 1: Use case diagram for the pedestrian navigation system

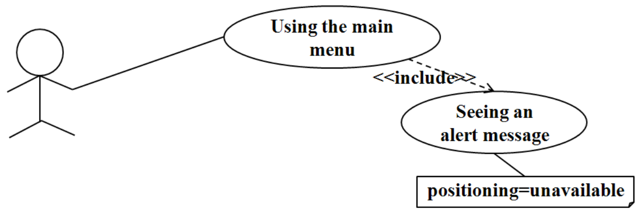

Figure 2: The context-dependent use case that displays an alert message

3.1 Abstract contexts

A context in our model is defined in terms of variables that take finite states (values). In 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." 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 several use cases. Figure 2 shows a 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].

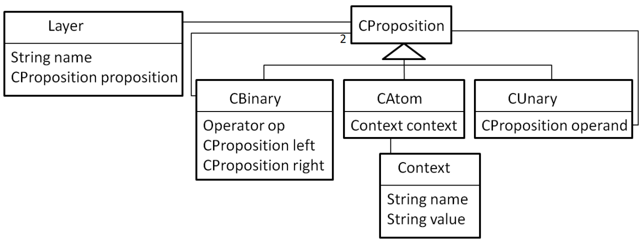

Figure 3: The model of layers

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 ∧ and ∨, `CUnary` represents the unary operator ¬, 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 lists 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:

- 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 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

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

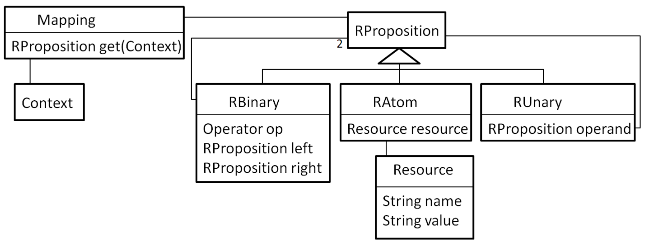

Figure 4: The model of resources and their mapping

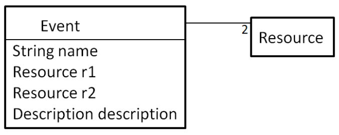

Figure 5: The model of events

(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 identifies 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 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.

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

```
class Navigation extends MapActivity
    implements Runnable, LocationListener {
  MyLocationOverlay overlay;
  void onStatusChanged(..) { .. }
  void run() {}
  void onCreate(Bundle status) {
    .. overlay.runOnFirstFix(this); ..
  }

  layer CityMap when StrongGPS || !FPExists {
    void run() { .. }
  }
  layer FloorPlan
    when !StrongGPS && WifiConnect && FPExists {
    void run() { .. }
  }
  layer StaticMap when !GPSon && !WifiConnect {
    void run() { .. }
  }
}
```

Figure 6: Layers and partial methods in EventCJ

4. MAPPING TO THE IMPLEMENTATION

This section discusses how the facts represented by our model are modularly mapped to each linguistic construct of the existing COP language, namely EventCJ [17, 18]. To make this paper self-contained, we firstly provide a short introduction to EventCJ. Then, we define the mapping from our model to EventCJ to demonstrate how the mapping 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` clauses in the layer declarations (this feature is available from the later version of EventCJ [18]), 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 left-most applicable rule is applied. Thus, the first rule above is read as, "upon the generation of `GPSEvent`, the layer `GPSon` is activated."

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 [20], 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.

[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.

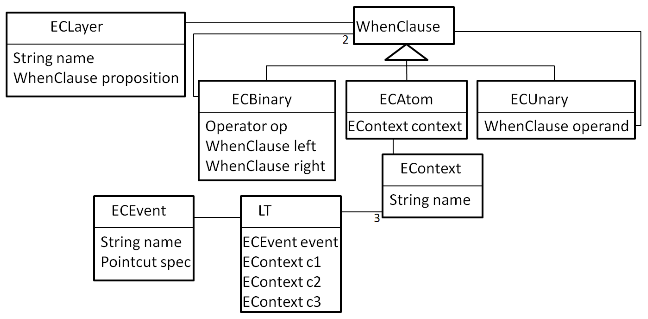

Figure 7: The model of EventCJ

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

We define the mapping 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`):

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

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`):

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

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`):

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

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 [16].

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:

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

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 [15], and we do not describe it in detail in this paper. The method for designing layers is a straightforward extension of [15].

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. RELATED WORK

Several COP languages have been developed thus far, and most of these share the same abstraction mechanism based on layers and partial methods [4, 5, 9, 13]. Thus, we can apply the same implementation scheme for these languages to translate context-dependent use cases into layers and partial methods. Since most of the existing COP languages are based on a dynamically scoped layer activation mechanism via so-called `with`-blocks, we need a different treatment to map context changes to the implementation; i.e., context changes and events specify the locations in source code where `with`-blocks should be inserted.

Considerable effort has been devoted to apply relatively recent programming paradigms, specifically aspect-oriented programming and feature-oriented programming, to software development. In particular, Jacobson's approach of aspect-oriented software development with use cases [16] is similar to our approach in many ways. For example, events declared in use case scenarios are based on the notion of extension points under Jacobson's approach. The major difference is that in our approach, events are not used as extension hooks but as triggers of layer transitions. Feature-oriented software development [3] is a method that maps feature diagrams [19], 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. Costanza proposed a method to analyze the dependency between layers using feature diagrams [8]. The objective of our approach is not to analyze such dependency between layers but to analyze dynamic context changes.

Context-aware applications may effectively be developed by the application of frameworks. Henrichsen and Indulska proposed a software engineering framework for pervasive computing [12]. It provides a modeling language called CML to model contexts based on four *types* and to describe states of contexts using propositional logic. The literature [7] describes a framework for Web applications that implement adaptation on the basis of contexts. The literature [6] describes an ontology-based framework for service selection depending on contexts. One drawback of framework-based solutions is that they are only applicable in specific application domains. Our approach may effectively be applied to wider range of application domains, because it does not depend on any special purpose frameworks.

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 facts represented by it is injectively mapped 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 a simple example. How to instantiate this model (i.e., 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, 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.

[6] 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.

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

[8] Pascal Costanza and Theo D'Hondt. Feature descriptions for context-oriented programming. In *2nd International Workshop on Dynamic Software Product Lines (DSPL'08)*, 2008.

[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] Patrick Eugster and K.R. Jayaran. EventJava: An extension of Java for event correlation. In *ECOOP'09*, volume 5653 of *LNCS*, pages 570–594, 2009.

[11] Vaidas Gasiunas, Lucas Satabin, Mira Mezini, Angel Nú nez, and Jacques Noyé. EScala: Modular event-driven object interactions in Scala. In *AOSD'11*, pages 227–240, 2011.

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

[13] 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.

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

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

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

[17] 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.

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

[19] 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.

[20] 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.

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

[22] 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.

[23] Angel Nú nez, Jacques Noyé, and Vaidas Gasiūnas. Declarative definition of contexts with polymorphic events. In *COP'09*, 2009.

[24] Hridesh Rajan and Gary T. Leavens. Ptolemy: A language with quantified, typed events. In *ECOOP'08*, pages 155–179, 2008.

[25] 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.

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

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

[28] Guido Salvaneschi, Carlo Ghezzi, and Matteo Pradella. ContextErlang: Introducing context-oriented programming in the actor model. In *AOSD'12*, 2012.

[29] 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.