Context-Oriented Software Development with Generalized Layer Activation Mechanism*

Tetsuo Kamina[1], Tomoyuki Aotani[2], Hidehiko Masuhara[2], and Tetsuo Tamai[3]

[1] Ritsumeikan University, 1-1-1 Noji-higashi, Kusatsu, Shiga, 525-8577, Japan
kamina@acm.org
[2] Tokyo Institute of Technology, 2-12-1 Ohokayama, Meguro, Tokyo, 152-8550, Japan
aotani@is.titech.ac.jp, masuhara@acm.org
[3] Hosei University, 3-7-2, Kajio-cho, Koganei, Tokyo, 184-8584, Japan
tamai@acm.org

Abstract. Linguistic constructs such as `if` statements, dynamic dispatches, dynamic deployments, and layers in context-oriented programming (COP) are used to implement context-dependent behavior. Use of such constructs significantly affects the modularity of the obtained implementation. While there are a number of cases where COP improves modularity related to variations of context-dependent behavior and controls of dynamic behavior changes, it is unclear when COP should be used in general.

This paper presents a study of our software development methodology, context-oriented software engineering (COSE), which is a use-case-driven software development methodology that guides us to specification of context-dependent requirements and design. We develop a systematic method to determine an appropriate linguistic mechanism to implement context-dependent behavior and dynamic changes in behavior during modeling and design, leading to the mechanized mapping from requirements and design artifacts formed by COSE to the COP implementation, which is demonstrated in our COP language ServalCJ through three case studies. We then identify the key linguistic constructs that make COSE effective by examining existing COP languages.

1 Introduction

Context awareness is a major concern in many application areas. Hirschfeld et al. define *context* as "(a piece of) information which is computationally accessible" [26]. However, this definition is too general to identify contexts; therefore, criteria for identification of contexts are necessary. For example, one important factor of context-awareness is a system's capability to behave appropriately with respect to surrounding contexts. Thus, a context is identified by observing behavioral changes in the application. An example of a context-aware application

* This paper is an extended version of our previous papers [31], [33]. In this paper, we refine principles and provide a more detailed description of our methodology. It also contains a case study not included in previous work.

is a ubiquitous computing application that behaves differently in different situations, such as different geographical locations, indoor or outdoor environments, or weathers. In this case, some specific states or situations are contexts. An adaptive user interface is also context aware as it provides different GUI components (behavior) depending on the current user task (context).

There are a number of constructs to implement context-dependent behavior, such as conditional branches using `if` statements, method dispatch in object-oriented programming (e.g., state design pattern), and dynamic deployment of aspects in aspect-oriented programming (AOP). Context-oriented programming (COP) [26] provides another mechanism to implement context-dependent behavior. This mechanism is often called a *layer*, which is a program unit comprising implementations of behavior that are executable only when some conditions hold.[4] In particular, COP provides disciplined activation mechanisms to ensure some consistency in dynamic changes in context-dependent behavior, such as scoping [7], model checking [29], dynamic checking of required interactions and constraints between different contexts [21], and a generalized layer activation mechanism [32]. Use of such constructs significantly affects the modularity of the obtained implementation, and research into COP shows a number of cases where COP can modularize variations of context-dependent behavior that are difficult to modularize using other approaches.

However, it is unclear when COP should be used in general. In particular, there are no systematic methodologies to determine a method to implement context-dependent behavior and the associated dynamic changes, which significantly affect software modularity and should be determined during modeling and design. Furthermore, there are no systematic methods to determine an appropriate activation mechanism to implement dynamic changes in behavior. A number of COP mechanisms have been proposed to date [7], [9], [17], [21–23], [25], [26], [29], [47], [51]. An appropriate mechanism must be selected from among them to implement a design artifact.

This paper presents a study of our software development methodology, context-oriented software engineering (COSE), that organizes the specifications of contexts and the dependent variations of behavior.[5] An overview of the COP development process, even if it is not in depth, can lead to further research on each stage of the development process. In particular, we answer the following research questions (RQs) based on this methodology.

RQ1. How should contexts and behavior depending on the contexts be elicited from the requirements?

[4] Some COP languages do not provide a linguistic construct to pack such implementations as a single unit. However, this is not a significant difference. In the remainder of this paper, we refer to a set of functions (methods) annotated with the same "conditions" for dispatch as a layer irrespective of whether they are packed into a single unit.

[5] This approach is based on Jacobson's object-oriented software engineering (OOSE) [27]. We refer to our approach as "COSE" to make its connection to OOSE clear.

RQ2. When should we apply COP rather than other development methods?
RQ3. How do COP mechanisms support predictable control of changes in context-dependent behavior?

We answer **RQ1** and **RQ2** by providing a systematic way to identify contexts and determine an implementation method for context-dependent behavior during modeling and design. This systematic approach is based on several principles, which are validated through three case studies. To answer **RQ3**, we provide a mechanized modular mapping from a specification developed by COSE to an implementation in the ServalCJ COP language[6]. ServalCJ provides a generalized layer activation mechanism that supports all existing COP mechanisms.

Methodology. Based on the use-case-driven approach [27], COSE represents the requirements for a context-aware application using contexts and context-dependent use cases. A context is represented in terms of Boolean variables that determine whether the system is in that context[7]. A context-dependent use case is a specialization of another use case that is applicable only under specific contexts. From these requirements, COSE derives a design model that can be translated into a modular implementation. This design method classifies variations of context-dependent behavior into those implemented by appropriate mechanisms, such as layers in COP and other traditional mechanisms, such as class hierarchies and `if` statements. This classification drives mechanized mapping from requirements to implementation. We selected ServalCJ as the implementation language because it provides a generalized layer activation mechanism, which, to the best of our knowledge, supports all existing COP mechanisms. This mapping ensures that each specification in the requirements is not scattered over multiple modules in the implementation, and each module is not entangled with multiple requirements.

Case Studies. We conducted three case studies of different context-aware applications to demonstrate the effectiveness of our approach. The first is a conference guide system, which serves as a guide for an academic conference, including management of an attendee's personal schedule, navigation help inside the venue and around the conference site, and a social networking service function, such as a Twitter client. The second is CJEdit, a program editor that provides different functionalities relative to cursor position. This example, which was first introduced by Appeltauer et al. [8], is a well-known COP application. The third is a maze-solving robot simulator that provides a number of variations of context-dependent behavior, such as adaptive user interfaces and adaptive robot behavior. In these case studies, we successfully organized context-related specifications

[6] `https://github.com/ServalCJ/pl`

[7] Keays also proposed COP [35], where a context is a named identifier (e.g., location) that identifies the type of *open terms* (holes in the code skeleton) that are filled at runtime with pieces of code corresponding to a specific context value (e.g., location in "Tokyo"). This paper is based on Hirschfeld's COP [26] where a context is represented as a *layer* that dynamically takes two states, i.e., active and inactive, and thus can be represented as a Boolean variable.

by applying COSE and directly mapped these specifications to their implementations in ServalCJ.

To examine existing language features and discuss the features that make the COSE methodology effective, we analyze linguistic constructs from several existing implementation techniques (including non-COP techniques). A notable finding is that, while most existing COP languages directly specify the execution point when the corresponding context becomes active, in the case studies there are a number of situations where the use of the implicit layer activation mechanism that indirectly specifies layer activation using conditional expressions would be preferred. Although currently the implicit layer activation mechanism may not function effectively, it can be an effective tool to independently implement the dynamic changes of behavior specified in the requirements.

Research Roadmap. Although the case studies indicate that our approach is promising, we also identify a number of interesting open issues, which comprise our future research roadmap. First, to address scattered context-dependent behavior in requirements of the system-to-be written in inconsistent formats, we plan to develop a systematic method to identify contexts. Second, our approach is based on use cases; however, it is also desirable to explore how similar approaches can be applied when use cases are not appropriate to analyze requirements. Third, we have identified issues in the evaluation of our methodology. Fourth, since there is a performance issue in the implicit layer activation, we plan to investigate the optimization of implicit activation. Furthermore, analyzing when event-based activation (i.e., the way in which the execution points where context activation occurs are explicitly represented) is expected to be useful and desirable. Finally, since the case studies used in this paper are standalone and conducted using a single language, it is also desirable to study how the approach can be applied to more sophisticated environments, e.g., distributed, multi-language environments.

Organization. The remainder of this paper is organized as follows. In Section 2, we identify the difficulties in the development of context-aware applications and discuss the limitations of existing approaches. In Section 3, we elaborate the research questions and list the principles that will be validated through the case studies. In Section 4, we illustrate the systematic organization of context-dependent requirements and their classification into those implemented by appropriate linguistic mechanisms. In Section 5, we provide mechanized mapping from the artifacts obtained by COSE to modular implementation in existing COP mechanisms. In Sections 6 and 7, we show other case studies and provide an informal evaluation of COSE using these case studies, respectively. Finally, Section 8 concludes the paper and presents our future research roadmap.

Differences from Previous Work. This paper is an extended version of our previous papers [31, 33]. In addition to the identification of context-dependent behavior whose activation should be controlled by COP mechanisms, this paper addresses the following issues:

Identification of the subject of a particular layer activation. While our previous papers implicitly considered that layer activation globally affects the whole application, this paper identifies other cases wherein layer activation affects a particular object or control-flow.

Selection of different layer activation mechanisms. This paper discusses the selection criteria for different layer activation mechanisms, which was not discussed in our previous papers. Existing layer activation mechanisms differ in scope and duration, and we should select the most suitable mechanism.

These issues are discussed using the maze-solving robot simulator case study, which also considers the case where a context-dependent use case appears in the top-level, thereby leading to refinement of principles.

2 Motivation

We explain the motivation to develop a new context-oriented software development methodology by introducing an example of a context-aware application and explaining the difficulties in the development of context-aware applications and the limitations of existing approaches.

2.1 Context-aware Application Example: Conference Guide System

The conference guide system serves as a guide for an academic conference. This system, which provides the conference program, schedule management, and navigation help inside the venue and around the conference site, is implemented on an Android smartphone. The guide system also has a Twitter that attendees can use to comment on talks presented at the conference. This system has several context-related behavioral variations.

- The conference program is provided online. The user can view the online program on an Android smartphone. The program is downloaded and cached in a local database in case the online version becomes unavailable. In the program, the user can select the sessions they will attend. The selected sessions are listed in the personal schedule. If sessions have not been selected, the listing cannot be accessed.
- The system provides a map function. When the user is within the conference venue, the map provides a floor plan of the venue. When the user is outside the venue, a map of the area around the conference site is provided. This area is updated when the user's position changes. Positioning is based on GPS or the Wi-Fi connection. If the system cannot determine whether the use is outdoors or indoors, it provides a static map of the area around the conference site.
- The system provides a Twitter client, which is available only when the Internet is available.

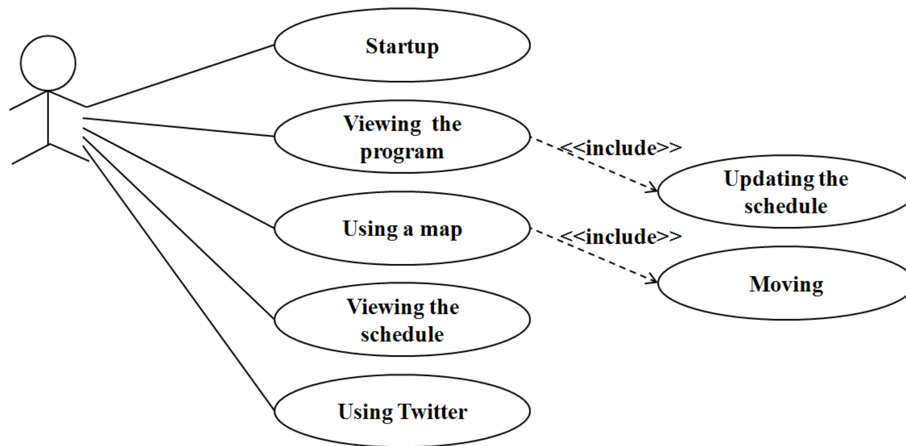


Fig. 1. Use case diagram for the conference guide system

A use case diagram for the conference guide system is shown in Fig. 1. In addition to the initial "Startup" use case, there are four use cases that involve user interactions. "Viewing the program" includes "Updating the schedule," i.e., the user selects sessions to attend, and "Using a map" includes "Moving," i.e., the user moves, and the new position is detected by the positioning system.

2.2 Difficulties

Even though this is a simple example, a number of difficulties in the development of context-aware applications can be identified. Before discussing these difficulties, we note that the definition of a context tends to be too general for identifying context-dependent behavior. Therefore, we first summarize three viewpoints that should be considered when identifying contexts.

Requirements Variability. A context-aware application changes behavior with respect to the currently executing context, i.e., there are a number of variations of behavior depending on the context. Thus, we need to identify contexts and the related requirements variability. For example, in the conference guide system, contexts such as outdoors, availability of the list of selected sessions, and availability of the Internet can be identified. However, identification of contexts is not trivial. After the identification of the outdoor context, it is unclear whether we should also identify the indoor context, or represent it by means of the outdoor context (i.e., !outdoors).

Different Levels of Abstraction. Contexts have different abstraction levels, and contexts at the abstract level consist of multiple concrete contexts. For example, the availability of positioning systems depends on hardware specifications, such as the availability of GPS and/or wireless LAN functions. Thus, we must define

contexts precisely in terms of the target machine. This multiple dependency leads to difficulty in defining precisely when the variation of behavior switches at runtime because there may be a number of state changes in the target machine that trigger a context change. Furthermore, some executing hardware states may barrier or guard the change of abstract contexts.

Multiple Dependencies among Contexts and Behavior. We must also analyze dependencies between contexts and variations of behavior carefully because some variations depend on multiple contexts. For example, in the conference guide system, if we identify outdoor and indoor situations as different contexts, the display of a static map is dependent on such contexts because this behavior is executable only when the system cannot determine whether the user is outdoors or indoors. Generally, multiple dependency depends on how we identify contexts, and multiple contexts may barrier or guard the execution of context-dependent behavior. This dependency becomes more complicated when we consider different levels of context abstraction.

We assume that contexts are identified based on these viewpoints. In our methodology, we address the following difficulties in the development of context-aware systems.

Proper Design and Implementation. We must select an appropriate method of design and implementation. In particular, a number of context-dependent requirements are volatile and crosscut multiple use cases. Therefore, such requirements require modular implementations to hide details that are likely to change [41].

Requirements Volatility in Context Specification. Technologies for sensing context changes are very complex. Such technologies evolve continually, which indicates that requirements specifications for context sensing are subject to change. For example, it seems initially appropriate to define the outdoor/indoor contexts based on the status of the GPS receiver. However, in future, this definition may need to be based on the status of air pressure sensors or other technologies, such as an RFID receiver, that are not currently implemented in smartphones.

Crosscutting of Contexts in Multiple Use Cases. In context-aware applications, a number of contexts are scattered over multiple use cases. For example, in the conference guide system, the conference program is downloaded through the Internet (to let the user access an up-to-date program) only when the Internet is available. Similarly, the availability of the Twitter client depends on the availability of the Internet. Thus, the context "the Internet is available" crosscuts two use cases, "Viewing the program" and "Using Twitter." A systematic way to determine such a situation and select the appropriate implementation mechanism for this specification is necessary.

Managing Dynamic Changes. Predictable control of contexts and behavioral changes is required, and this is a challenge for the following reasons.

Crosscutting of Behavior Changes. One of the most important properties of context-aware applications is that they change behavior at runtime. Thus, we need to identify when a behavior variation switches to another variation. However, as discussed above, a behavior variation may depend on multiple (abstract) contexts, and each context may depend on a number of concrete contexts. Furthermore, changes of such concrete contexts are scattered over the execution of the application. Since context specifications are subject to change, it is desirable to encapsulate them.

Interferences Between Behaviors. A situation wherein dynamically activated behaviors can interfere with other behaviors may occur. Such interference may result in unpredictable behavior and should be avoided. Determining whether such interference exists is also a challenge in the development of context-aware systems.

Translation to Modular Implementation. We must carefully trace which requirements are implemented by which modules. It is also desirable that a module in the implementation only serves a single requirement and is not entangled by several requirements. Thus, to support modularity, it is desirable that there be injective mapping from the specification to the implementation.

2.3 Problems in Existing Approaches

Although there have been intensive research efforts to improve each stage of development of context-aware applications, few attempts have been made to develop a methodology to organize the whole development process.

A number of COP languages have been developed. A family of COP languages provides a linguistic construct called a layer to pack related context-dependent behavior into a single module [7], [9], [17], [25]. Other COP languages emphasize representing the dependency between contexts [21, 22] and do not provide "layers," though, in this paper, this difference is not significant, and we refer to a set of functions (methods) annotated with the same "contexts" as a layer, irrespective of whether they are packed into a single module.

Compared to research into programming languages, little research effort has been devoted to systematizing the design of context-oriented programs. For example, the process of discovering layers from requirements is unclear. Determining when the use of layers is preferable to the use of existing object-oriented mechanisms and `if` statements in order to implement context-dependent behavior also remains unclear. Cardozo et al. proposed the feature clouds programming model [14], which equates layers (i.e., "behavioral adaptations") to features in feature-oriented software development (FOSD) [4]. Although this model clarifies correspondence between features and COP and advances the feature model by introducing dynamic adaptation of features by means of COP mechanisms, it does not clarify the process of discovering contexts and layers. Context petri nets (CoPNs) [13] were proposed to formalize the semantics of COP, in particular

the semantics of multiple context activations in Subjective-C [21]. A tool based on CoPNs was developed to analyze consistency in the activation of contexts. However, this tool does not target the discovery of contexts and layers.

There are a number of software development methodologies. Object-oriented methodologies are useful for discovering objects and classes through requirements analysis. Aspect-oriented software development (AOSD) methodologies [28], [43] are useful for determining and modularizing crosscutting concerns. FOSD [4] maps feature diagrams [34] to implementation. Feature diagrams, which are obtained by analyzing the software to be developed, are useful for analyzing dependencies among the features from which the software is constructed. Even though these methodologies provide a good starting point to consider how to develop context-aware applications, they do not focus on solutions for the aforementioned difficulties. We must extend existing methodologies to identify contexts and dependent behavior systematically to provide predictable control of changes in context-dependent behavior.

Recently, a number of approaches to discover, analyze, and implement contexts and variations of dependent behavior have been investigated. A number of requirements engineering methods [3], [19], [38, 39], [45, 46], [49] primarily focus on the discovery and analysis of (abstract) contexts and the variations of behavior that depend on them. These requirements engineering methods do not provide systematic ways to select implementation mechanisms for context-dependent behavior. Henrichsen and Indulska proposed a software engineering framework for pervasive computing [24]. However, they did not provide systematic ways to manage volatile requirements for concrete levels of context and implement them modularly. Specifically, they did not identify a set of variations that comprises a single module. Frameworks and libraries for context-aware applications provide context-aware software components and thus enhance reusability, which addresses some of the difficulties mentioned above [1], [12], [15], [44]. However, such frameworks and libraries are domain specific, and few general solutions for context-aware applications are provided.

3 Research Questions and Principles

To organize the software development methodology for context-aware applications, we provide three research questions. To answer these questions, we also list three principles and validate them through case studies.

3.1 Research Questions

We answer the following research questions.

RQ1. How should contexts and behavior depending on the contexts be elicited from the requirements?

This research question, which has been partially answered by existing approaches, is the most fundamental. To implement context-dependent behavior,

we must first determine what the contexts are. However, this identification of contexts should be organized as input for a decision about the design of modules made in the subsequent development stage. The existing requirements engineering methods explained above do not have such a concern, and we provide the answer for **RQ1** from this perspective.

RQ2. When should we apply COP rather than other development methods?

There are a number of linguistic constructs to implement context-dependent behavior, and we must select the appropriate construct to realize better modularity. In particular, existing work into COP has not provided an answer for when we should use COP.

RQ3. How do COP mechanisms support predictable control of changes in context-dependent behavior?

A number of COP mechanisms, in particular *layer activation mechanisms*, have been proposed, and they support predictable control of behavioral changes with respect to context changes by, e.g., preventing us from simultaneously activating conflicting layers. Each mechanism has its own advantages and disadvantages, and there are no methods to select the most appropriate mechanism.

3.2 Principles

As the first step to provide the answers for these research questions, we identify the following principles to identify contexts and context-dependent behavior. We then developed the context-oriented software development methodology, COSE, which is explained in the following sections.

Principle 1 *Factors that exist outside a particular unit of computation and dynamically change the behavior of that unit are candidates for contexts.*

A context is a factor that changes the behavior of something on which we focus. Thus, to identify contexts, it is good to begin by looking for factors that change the behavior of such a "something," which is also identifiable under the specific computation model (e.g., an object in OO programming languages or a function in functional programming languages).

Note that this principle is revised from a previous description [33] by considering the *unit of computation* that is affected by the contexts. This consideration leads to proper selection of layer activation mechanisms in the implementation (e.g., selecting global activation rather than per-instance activation). Further discussion is provided in Section 4.

Principle 2 *Each factor that dynamically changes the system's behavior is represented as a variable, and an activation condition for a context that determines whether the system is in that context is a logical formula comprising those variables.*

In many cases, a factor that changes system behavior has only two states. For example, whether a user is outdoors has only two states, yes or no. The availability of a network also has two states, available or unavailable. Battery level can also have two states, low or high. Each of these factors can be represented as a Boolean variable.

Sometimes, such a factor is composite, which implies that a context can be represented as a logical formula comprising a set of factors that can be represented as a Boolean variable. This principle fits well with existing COP languages wherein a context consists of subcontexts, such as in Subjective-C [21], or a layer activation is triggered by a composition of other layer activations [16], [30].

In some cases, such factors may have more than two states. For example, a location may take a number of values such as "Tokyo," "Lugano," etc. In such cases, we consider each value as a context. For example, we consider the context "whether the user is in Tokyo." This may result in quite a large number of contexts (e.g., we may list thousands of cities), and it is difficult to prepare such a listing. Generally, COP requires pre-listing of behavior variations, and a large number of contexts are unlikely to be modularized using COP but can be implemented using other techniques, such as abstraction over parameters.

In some COP languages like Subjective-C, a context is not a Boolean but has an actual activation count. The above principle does not rule out such languages. A context is identified in terms of an activation condition that is a trigger for the context-dependent behavior. This activation condition should be Boolean even when a context has an actual activation count.

Principle 3 *If multiple variations of context-dependent behavior share the same context and variations are not specializations*[8] *of the same behavior, they should be implemented using a layer.*

This principle explains the situation wherein the same context is scattered over a number of behavioral variations in the system. A layer in COP can modularize such crosscutting behavior. In contrast, if the context affects only a single behavior variation or such variations are a specialization of the same behavior, we may also consider other implementation mechanisms, such as `if` statements and method dispatching in object-oriented programming.

4 Context-Dependent Requirements and Design

We propose COSE, a use-case-based methodology for context-oriented software engineering. It represents the requirements for a context-aware application using contexts and context-dependent use cases. A context is represented as a Boolean formula that represents whether the system is in that context. A context-dependent use case is a specialization of another use case applicable only in some specific contexts.

[8] By "specialization," we mean a specialization relationship that appears in class and use case hierarchies, i.e., a specialization consists of more details than its parent.

Based on this requirements model, COSE further derives a design model that can be translated into a modular implementation (Section 5). COSE is based on the use-case-driven approach. It provides a systematic mapping from context-dependent use cases to modules provided by existing COP languages, i.e., *layers*, just as the AOSD method proposed by Jacobson where each use case is implemented using an aspect [28]. Our design method classifies context-dependent behavior variations into those implemented by appropriate implementation mechanisms, such as layers in COP, and those implemented by other traditional mechanisms, such as class hierarchies and `if` statements. We identify the following design constituents.

1. Groups of context-dependent use cases, each of which share the same contexts. Context-dependent use cases in the same group simultaneously become applicable when the contexts hold. To modularize dynamic behavioral changes, they should be modularized into a layer in COP languages.
2. Classes participating in use cases by applying the standard use-case-driven approach.
3. Detailed specification of contexts based on the identified classes and frameworks on which the system depends.

In the following sections, we overview each step of COSE using the conference guide system example introduced in Section 2.

4.1 Identifying Contexts and Context-Dependent Use Cases

The first step of COSE is to identify contexts and context-dependent use cases. We extend the original use-case-driven method [27] with context-dependent use cases that are applicable only in specific contexts. By observing use cases, we can see that a number of behavior variations of some units of the system exist with respect to some outside conditions, which are subject to change at runtime. As mentioned in Principle 1, such conditions are candidates for the variables that determine the current context. For example, in the conference guide system, we identify the use case "Startup" where the user starts the system. This use case includes the behavior of the whole application initializing several parts of the system, and has two sub-use-cases, i.e., "Startup scheduler" (prepares the menu for the user's schedule) and "Startup Twitter" (prepares the menu for the Twitter client). All these sub-use-cases are applicable only when some conditions hold, such as the availability of the user's schedule and availability of the Internet. We can identify these conditions as candidates for contexts that change the behavior of the application. In the remainder of this paper, we refer to a context that changes the behavior of the entire application as a *global context*.

Another example is the "Using a map" use case, which is specialized to three use cases: "Using a city map," "Using the floor plan," and "Using a static map." These are applicable when the user is outdoors, when the user is indoors, and when the system cannot determine the user's location, respectively. Again, these specializations affect the whole application; thus, these user's situations are also candidates for global contexts.

Table 1. Listing of variables: the first stage

subject	name	description
global	hasSchedule	the user has registered at least one session or not
	hasNetwork	the Internet is available or not
	outdoors	the situation is outdoors or not
	hasPositioning	the positioning systems are available or not
	batteryLow	the battery level is low or not

Table 2. Refined listing of variables

subject	name	description
global	hasSchedule	the user has registered at least one session or not
	hasNetwork	the Internet is available or not
	outdoors	the situation is outdoors or not
	indoors	the situation is indoors or not
	batteryLow	the battery level is low or not

Generally, a context in our model is defined in terms of a set of Boolean variables that represents the condition of the subject of the behavior. We list the candidates for variables in the conference guide system in Table 1. In this table, we represent the subject (the whole application) as global. Note that this is the very early stage of listing candidates for variables that are directly observable from the behavior of the system-to-be, and we introduce one important criterion used to refine this listing.

> Each variable should not depend on other variables because such dependencies imply that a variable can be represented in terms of others.

A context should consist of a set of orthogonal variables; if they are not orthogonal, they should be exclusive. This criterion is required to keep the conditions constructed by these variables simple and ensure the completeness of contexts. Intuitively, being orthogonal means that every combination of values is possible. For example, in Table 1, hasSchedule, hasNetwork, outdoors, and batteryLow are orthogonal. However, outdoors and hasPositioning are not orthogonal because the combination outdoors && !hasPositioning is impossible (we assume that the conference guide system determines the outdoors situation using positioning systems). If it is not possible to represent such variables using just one single Boolean variable, then we should reformulate them as exclusive variables, which help analyze dependencies between layers. Thus, the variables outdoors and hasPositioning are divided into three exclusive variables representing outdoors, indoors, and no positioning is available, and the final one is exactly the case where the system cannot determine whether it is outdoors or indoors. The refined listing of variables is shown in Table 2.

Note that, as discussed in Section 2, requirements for context changes are often volatile. Thus, at this stage, it is preferable to keep contexts abstract to be prepared for future requirements changes.

Table 3. Use cases for the conference guide system

name	activation condition
Startup	
Startup scheduler	hasSchedule
Startup Twitter	hasNetwork
Viewing the program	
Viewing the online program	hasNetwork
Updating the schedule	
Using a map	
Using a city map	outdoors
Using the floor plan	indoors
Using a static map	!outdoors && !indoors
Moving	
Moving when outdoors	outdoors
Viewing the schedule	hasSchedule
Using Twitter	hasNetwork
Updating timeline frequently	!batteryLow
Updating timeline infrequently	batteryLow

A context-dependent use case is annotated with a logical formula that consists of the set of variables identified above. We call this formula an *activation condition* for that use case. Context-dependent use cases for the conference guide system are summarized in Table 3. The names of use cases are listed in the left column, and activation conditions that represent when the use case is applicable are listed in the right column. A name with an indent represents that this use case is a specialization of the use case listed in the above row in italics. A use case with an empty condition is context independent.

4.2 Grouping Context-Dependent Use Cases

A situation where multiple use cases are applicable in the same context implies that the context-dependent behavior is scattered over those use cases. To modularize dynamic behavioral changes, these context-dependent use cases should be grouped into a single module that is enabled (activated) when the condition holds and disabled (deactivated) when the condition does not hold. This is the situation Principle 3 explains, which is rephrased in terms of the use case driven method as follows. *If multiple context-dependent use cases that are not specializations of the same use case share the same context, their behavior should be implemented by using a layer.*

Table 4 lists the groups of context-dependent use cases. We can see that three contexts, i.e., hasSchedule, hasNetwork, and outdoors, are assigned to multiple context-dependent use cases. Thus, these use cases are grouped into a layer. We rename such contexts by capitalizing the first character (e.g.,`HasSchedule`, `HasNetwork`, and `Outdoors`), following naming traditions for layers in COP languages.

Table 4. Groups of context-dependent use cases

activation condition	use case
hasSchedule	Startup scheduler
	Viewing the scheduler
hasNetwork	Startup Twitter
	Viewing the online program
	Using Twitter
outdoors	Using a city map
	Moving when outdoors
indoors	Using the floor plan
!outdoors && !indoors	Using a static map
hasNetwork && !batteryLow	Updating timeline frequently
hasNetwork && batteryLow	Updating timeline infrequently

Table 5. Classes for each layer

layer	classes	position
`HasSchedule`	`MainActivity`, `Schedule`	class-in-layer
`HasNetwork`	`MainActivity`, `Program`, `Twitter`	class-in-layer
`Outdoors`	`Map`	layer-in-class
`Indoors`	`Map`	layer-in-class
`StaticMap`	`Map`	layer-in-class

We must also consider how to treat the remaining context-dependent use cases. Even though they do not share the condition with other use cases, some still have a relationship with other layers in that a subterm of their condition is the condition that activates the layer. For example, the condition for "Using a static map" includes the subterm outdoors, which is the condition that activates the layer `Outdoors`. To control dynamic behavior changes uniformly, activation of "Using a static map" should be managed in the same way as `Outdoors`. Thus, we also identify the context-dependent use case "Using a static map" as a layer, i.e., `StaticMap`. Similarly, we identify the context-dependent use case "Using the floor plan" as a layer, i.e., `Indoors`.

So far, we have identified at least five layers. We do not identify other use cases, e.g., "Updating timeline frequently" and "Updating timeline infrequently," as context-dependent. They are conceptually the same as alternative use cases, and the behavior variations should be so local that each of them can be implemented within a single class. Thus, they can be implemented by traditional OO mechanisms, such as inheritance and `if` statements.

4.3 Designing Classes

Each layer in COP consists of (partial) definitions of classes. By extending the original use-case-driven approach in a straightforward manner, we can identify classes and methods that participate in each layer.

First, from use case scenarios, we identify the names of classes. Since this is a straightforward adaptation of the original use-case-driven approach, we do not describe the details but briefly illustrate the result. Since the conference guide system is an Android application, each view of the application should be implemented as a subclass of the `android.app.Activity` class from the Android SDK framework[9]. The use case "*Startup*" identifies the `MainActivity` class, which will implement the main view of the application. Similarly, in the use cases "*Viewing the program*," "*Using a map*," "*Viewing the schedule*," and "*Using Twitter*," we identify an `Activity` class for each, i.e., `Program`, `Map`, `Schedule`, and `Twitter`, respectively. There are some other helper classes; however, only the `Activity` classes participate in the context-dependent behavior.

Table 5 summarizes this assignment of classes for each layer. While the layers `HasSchedule` and `HasNetwork` consist of multiple classes, other layers consist of just the `Map` class. This table also shows the preferred ways to allocate layers. There are two alternative ways to allocate layers, i.e., the class-in-layer style allocates the (partial) classes that implement the context-dependent behavior in the layer, and the layer-in-class style allocates the layer within the class. When a layer is scattered over several classes, the class-in-layer style is preferable. When a class is scattered over several layers, the layer-in-class style is better. Note that some COP languages support only one style [6]. In this case, we must conform to the style provided by the implementing language.

4.4 Designing Detailed Specification of Contexts

After designing classes, we can determine a more concrete representation of the contexts. While it is desirable to keep contexts abstract to allow changes in the details, we must also derive information about how they should be implemented. In particular, there are a number of layer activation mechanisms, and we must select an appropriate mechanism. Furthermore, as explained later, specifications for some contexts are complex; thus, we must identify more granular contexts that comprise the specified context.

Section 4.1 defines that the context hasSchedule holds when the user has selected at least one session to attend from the conference program. In terms of the Android SDK framework, this is represented as "a query on the `SQLite` instance returns at least one result." Thus, we define when the layer `HasSchedule` becomes active as follows, which is read as "the `getCount` method on the result of a `query` on an `SQLite` instance (i.e., `db`) returns an integer value that is greater than 0."

```
HasSchedule(SQLite db) :: db.query(..).getCount() > 0
```

Similarly, by inspecting Android SDK framework specifications, we can define when the layer `HasNetwork` becomes active as "the result of the call of the `getDetailedState` method on the result of the call of `getActiveNetworkInfo` on a `ConnectivityManager` instance (i.e., `cm`) is equal to the result of the access to the static field `NetworkInfo.DetailedState.CONNECTED`."

[9] `http://developer.android.com/sdk/`

```
HasNetwork(ConnectivityManager cm) ::
  cm.getActiveNetworkInfo().getDetailedState() ==
    NetworkInfo.DetailedState.CONNECTED
```

The cases for the outdoors and indoors contexts are more complex. They are affected by multiple states of the running machine. First, to determine whether the user is outdoors, the GPS device should be available. Second, the conference guide system determines whether the user is in the conference venue using the SSID of the connecting wireless LAN, which means that the wireless LAN connection should be available. Thus, activation of the `Outdoors` and `Indoors` layers is determined in terms of more fine-grained contexts.

```
Outdoors :: !WifiAvailable && GPSAvailable
Indoors :: WifiAvailable
```

In other words, `Outdoors` and `Indoors` are composite layers [30].

The context `WifiAvailable` is defined as follows assuming that `isWifiConnected` is an application method that returns `true` when the wireless LAN is connected and its SSID is some pre-defined value.

```
WifiAvailable :: Config.isWifiConnected()==true
```

The context `GPSAvailable` is defined as follows using the `isProviderEnabled` method provided by the framework.

```
GPSAvailable :: LocationManager.isProviderEnabled(
    LocationManager.GPS_PROVIDER) == true
```

All these concrete representations of contexts reveal that they are conditionals and can be implemented directly using a layer activation mechanism triggered by conditionals.[10] In Section 6.1, we show cases where other activation mechanisms are selected.

5 Mapping to Implementation

This section demonstrates how the design artifacts developed by COSE are systematically translated into a program with existing COP mechanisms. Generally, a layer identified in the previous section is implemented using a corresponding mechanism provided by the COP language chosen as an implementation language, such as a layer in ContextJ, a set of methods that share the same context in Subjective-C, or a context trait [23]. The detailed specification of contexts is then mapped to the corresponding layer activation mechanisms provided by that language.

In this paper, we choose ServalCJ [32] (a successor of EventCJ [29]) as the implementation language because it provides a generalized layer activation mechanism that supports most existing COP mechanisms. A context in ServalCJ is

[10] For COP languages that do not provide layer activation by conditionals, we must provide a workaround to implement such conditionals.

defined as a term of temporal logic with a call stack, which can represent most existing layer activation mechanisms. For example, it can specify two events, one of which activates the corresponding context and the other which deactivates that context (as in EventCJ's event-based layer transition). ServalCJ can also specify a control flow under which the corresponding context is active (as in JCop [9]). ServalCJ can select the target where such context specifications are applied, and that target can be a set of objects (*per-instance* activation) or the whole application (*global* activation). Furthermore, ServalCJ supports *implicit activation*, where activation of a context is indirectly specified using a conditional expression. As shown in the following sections, our methodology clarifies that this mechanism is notably useful for modular implementation.

A ServalCJ program comprises a set of classes, layers, and *context groups* where dynamic layer activation and the target for this activation are specified. The layers and classes identified in Sections 4.2 and 4.3 are implemented directly in layers and classes in ServalCJ, and the context specifications in Section 4.4 are implemented directly in context groups in ServalCJ. We explain the details in the following sections.

5.1 Implementing Layers

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

Fig. 2 shows an example of layers and partial methods in ServalCJ for the main view of the conference guide system. The `MainActivity` class extends the `Activity` class provided by the Android SDK framework and overrides the `onResume` method, which is called from the framework when this view resumes the execution. This method displays the main menu of the conference guide system as buttons for viewing the conference program and using the map. `MainActivity` also declares two layers, i.e., `HasSchedule` and `HasNetwork`. These layers define the context-dependent behavior of `MainActivity`[11]. `HasSchedule` defines the behavior when there is at least one session that the user would like to attend, and `HasNetwork` defines the behavior when the Internet is available. These layers extend the original behavior of `onResume` by declaring *after* partial methods, which are executed just after the execution of the original method when the respective layer is active[12]. For example, when `HasSchedule` is active, `onResume` also displays the menu button to check the user's schedule.

[11] Although Table 5 shows that it is preferable to implement these layers in the class-in-layer style, in Fig. 2, they are implemented in the layer-in-class style because ServalCJ currently only supports this style.

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

```
class MainActivity extends Activity implements View.OnClickListener {
  private GridLayout layout;

  @Override
  protected void onResume() {
    super.onResume();
    layout = new GridLayout(this);
    layout.addView(makeMenu("program", "Program"));
    layout.addView(makeMenu("map", "Map"));
  }

  private Button makeMenu(String tag, String label) {
    ..
  }

  layer HasSchedule {
    after protected void onResume() {
      layout.addView(makeMenu("schedule", "Schedule"));
    }
  }
  layer HasNetwork {
    after protected void onResume() {
      layout.addView(makeMenu("twitter", "Twitter"));
    }
  }
}
```

Fig. 2. Layers and partial methods in ServalCJ

5.2 Implementing Layer Activation

In COP languages, layers can be activated and deactivated dynamically, and ServalCJ provides declarative ways to perform such layer operations. These declarations are obtained directly from the design of detailed contexts (Section 4.4).

First, detailed context definitions are grouped based on the variables and contexts to which these definitions refer. We refer to such group as a *context group*. For example, `HasNetwork` refers to an instance of `ConnectivityManager` (and this is the only context that refers to that instance); thus, this context definition makes up one context group.

Fig. 3 shows a context group that is responsible for activating `HasNetwork`. The first line specifies the name of the context group, i.e., `Network`, followed by a specification of how this context group is instantiated. Since the context `HasNetwork` is identified as global (Section 4), this context group is declared as global, as specified by the modifier `global`. In line 2, the `perthis` clause specifies that the instance of `Network` is associated with an instance of `ConnectivityManager` (as specified using the `this` pointcut), which can be referenced through the variable `cm`.

```
global contextgroup Network(ConnectivityManager cm)
    perthis(this(ConnectivityManager)) {
  activate HasNetwork if(
    cm.getActiveNetworkInfo().getDetailedState()
      ==NetworkInfo.DetailedState.CONNECTED);
}
```

Fig. 3. Context group responsible for activation of `HasNetwork`

```
global contextgroup Schedule(MainActivity main)
    perthis(this(MainActivity)) {
  activate HasSchedule if(main.scheduleCounter > 0);
}
```

Fig. 4. Context group responsible for activation of `HasSchedule`

Line 3 declares when the layer `HasNetwork` is active using an *activate declaration*. A `when` clause specifies the condition when the layer is active. There are a number of ways to specify this condition, e.g., specify the join points where that context becomes active and inactive, specify the control flow under which that context is active, and specify the condition when that context is active. In Fig. 3, we use the `if` expression to specify the condition. With the `if` expression, we can use any Boolean-type Java expression. In this case, we simply copy the expression from the definition in Section 4.4.

We can declare a context group for `HasSchedule` in a similar way. One subtle issue is that the definition of `HasSchedule` contains an expression that requires local database access. If the developer has performance concerns, this definition is not preferred because this condition is tested at every call of the layered method (i.e., a method that consists of a set of partial methods) in ServalCJ. In our case, the definition of `HasSchedule` is refined to access the counter variable that is introduced to `MainActivity` and updated when the local database is updated as follows.

```
HasSchedule(MainActivity main) :: main.scheduleCounter > 0
```

The definition of the context group for `HasSchedule` is shown in Fig. 4.

The remaining layers are `Outdoors`, `Indoors`, and `StaticMap`. Since they share the same set of context references, they are grouped into one context group, which is shown in Fig. 5. Since this context group does not refer to any instance variables, it specifies no `perthis` and `pertarget` clauses. This context group is a *singleton*, i.e., it is created when execution initializes and remains until the application terminates.

Lines 2 and 3 define the *named contexts* `WifiAvailable` and `GPSAvailable`, which make it possible to refer to the activation conditions from several activate declarations that are used to specify when `Outdoors` and `Indoors` are active.

```
global contextgroup Situation {
  context WifiAvailable is if(Config.isWifiConnected()==true);
  context GPSAvailable is if(LocationManager.isProviderEnabled(
      LocationManager.GPS_PROVIDER)==true);
  activate Outdoors when !WifiAvailable && when GPSAvailable;
  activate Indoors when WifiAvaileble;
  activate StaticMap when !Outdoors && when !Indoors;
```

Fig. 5. Context group responsible for activation of `Outdoors`, `Indoors`, and `StaticMap`

The conditions declared by these named contexts are directly obtained from the definitions given in Section 4.4. Activate declarations for `Outdoors`, `Indoors`, and `StaticMap` are also obtained from the definitions given in Section 4.4. Note that we can use the logical operators `||`, `&&`, and `!` to compose propositions in the `when` clauses.

6 Other Case Studies

This section demonstrates two other case studies using COSE. The first is conducted to investigate the applicability of COSE to an existing, well-known COP application. The second study shows more interesting cases, which are not discussed in our previous paper [33], where different activation mechanisms with respect to the scope and duration are applied. Among these mechanisms, we should select the most suitable mechanism.

The first case study develops the CJEdit program editor that is first implemented by Appeltauer et al. [8]. Since the original implementation of CJEdit exists, we do not perform this case study from scratch. Instead, we use the original implementation as a prototype for this case study. This case study shows that COSE is applicable to the development of a well-known COP application. Details of this case study are found in our previous paper [33].

The second case study develops a maze-solving robot simulator [32] wherein a context can be per-instance, per-control-flow, and global and the events triggering the context changes are explicit. The second case study also shows the situation where a context-dependent use case is *not* a specialization of another use case. Principle 3 assumes that there is a *default* use case for context-dependent use cases; however this assumption does not hold in this case study. The following principle compensates for such a situation.

Principle 4 *If a top-level use case is also context-dependent, it should be implemented using a layer.*

Generally, multiple objects can participate in a use case and, as Jacobson suggests [28], behavior in such a use case crosscuts multiple objects and thus should be implemented using a layer.

6.1 Maze-solving Robot Simulator

This application simulates how a line-tracing robot solves a maze[13]. The robot solves a maze comprising black lines on a sheet of white paper. After solving the maze, the robot runs the optimized path from the starting point to the goal. The maze-solving phase comprises the following behavior.

- Performing line tracing until the robot reaches an intersection, a corner, or a dead-end (we refer to these as *intersections* for simplicity).
- Detecting intersections using reflectance sensors attached to the robot.
- Making a turn at each intersection according to the implemented algorithm, e.g., left-hand rule, right-hand rule, and Trémaux's algorithm[14]. In this example, we set the left-hand rule as default behavior.
- Remembering the sequence of behavior at each intersection (and possibly all visited intersections) and calculating the optimized path from the starting point to the current intersection by eliminating dead-ends.

The robot can also display some debugging information, such as currently visited paths, on the small display attached to the robot.

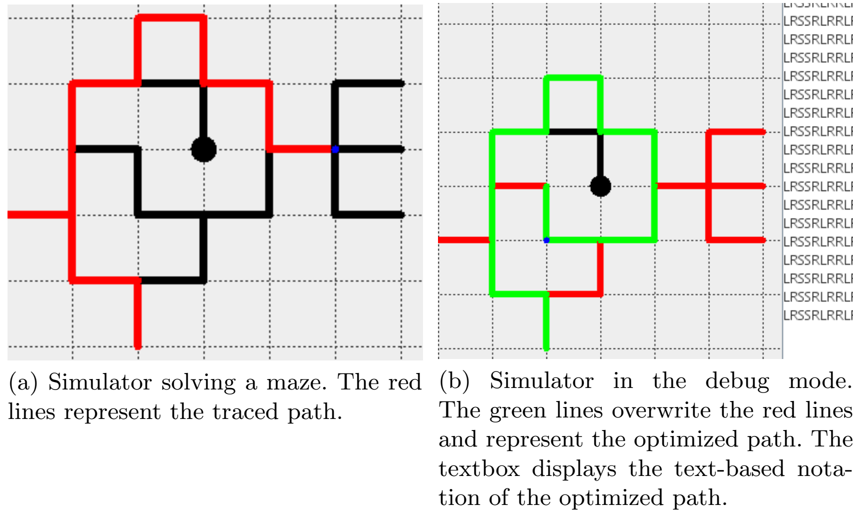
The simulator emulates the behavior of the maze-solving robot. In this simulator, the maze is modeled as a graph where each node represents an intersection that provides coordinates to represent its position. An instance of the robot emulates maze-solving on this model, e.g., line-tracing is simplified by updating the current position of the robot according to the destination of the selected edge, which models the segment. Here, a segment is a part of a path from one intersection to another.

For the user, this simulator provides a number of functionalities: editing a maze, simulating how the robot solves the maze, and simulating how the robot follows the optimized path after solving the maze. These functionalities are exclusive, i.e., when we are editing a maze, we cannot run any simulations for solving the maze and running the optimized path, and so on. These functionalities are switched when the user finishes editing the maze (or loads the pre-edited maze) and when the robot finishes solving the maze. The simulator provides GUI tools, such as a menu bar and menu buttons that are automatically switched when the functionalities are switched. During maze-solving, the visited intersections and segments are colored to visualize the traced path (Fig. 6 (a)). Furthermore, while the robot is solving the maze, the user can select the debug mode. In debug mode, the color of intersections and segments in the currently calculated optimized path are changed and text based notation of the optimized path is displayed (Fig. 6 (b)).

From this description of the behavior, we first derive the variables that determine the current context (Table 6). Unlike the previous examples, in this

[13] This simulator is inspired by a real maze-solving robot (Pololu 3pi Robot: `http://www.pololu.com/product/975`) and the behavior of the simulator is modeled by following the sample program provided by the 3pi Robot distribution.

[14] Only Trémaux's algorithm can solve the maze with loops.



(a) Simulator solving a maze. The red lines represent the traced path.

(b) Simulator in the debug mode. The green lines overwrite the red lines and represent the optimized path. The textbox displays the text-based notation of the optimized path.

Fig. 6. The maze-solving simulator. The lines indicate paths within the maze. The start is the lowermost dead end. The black circle represents the goal.

Table 6. Maze-solving simulator variables

subject	**name**	**description**
global	editingMaze	The user is editing a maze
	solvingMaze	The user is simulating maze-solving
	runningMaze	The user is simulating running the solved maze
	debugging	Showing the debugging information
robot	rightHand	Solving maze using the right-hand rule
	tremaux	Solving maze using the Trémaux algorithm
cflow	displaying	Displaying the path information

case we identify three types of subjects. From the perspective of the user, this simulator provides different functionalities with respect to the current user task. These functionalities are identified as global because the user is not aware of a particular part of the system. Several maze-solving algorithms are selected dynamically. These algorithms are executed by an instance of the robot modeled in the simulator. Thus, we identify the subject for these algorithms as a robot. Finally, we identify the context that holds when path information is displayed. Path information comprises a particular control flow; therefore, we identify its subject as a control flow (cflow).

Table 7 lists context-dependent use cases for the maze-solving simulator. First, we identify four top-level use cases, i.e., "*Editing a maze*," "*Solving a maze*," "*Running a maze*," and "*Debugging*." For "*Solving a maze*," we further derive context-aware alternatives based on the selected algorithm. "*Debugging*" also includes displaying the debugging information, which is executable only within the control flow of the displaying behavior.

Table 7. Use cases for maze-solving simulator

name	**activation condition**
Editing a maze	editingMaze
Solving a maze	solvingMaze
Solving with right-hand rule	rightHand
Solving with Trémaux	tremaux
Running a maze	runningMaze
Debugging	debugging
Displaying debug info.	debugging && displaying

Table 8. Classes for each layer of maze-solving simulator

layer	**classes**
`EditingMaze`	`View`
`SolvingMaze`	`Robot`, `View`
`RunningMaze`	`Robot`, `View`
`Debugging`	`Robot`, `View`
`UnderDebugging`	`Segment`, `Intersection`

As suggested by Principle 4, we first identify the top-level use cases as layers, i.e., `EditingMaze`, `SolvingMaze`, `RunningMaze`, and `Debugging`. "Displaying debug info." is a use case included by "Debugging," but it is not a specialization of other use cases; therefore, we identify it as a layer, i.e., `UnderDebugging`. By following COSE, the remaining two use cases are not identified as layers here. However, in a later step we actually come to consider that they are layers.

Next, we identify the names of classes from the use case scenarios. Table 8 lists the important classes for implementing context-dependent behavior. The class `Robot` models the behavior of the virtual robot, and the class `View` provides the view for the user. Many of the layers crosscut both classes. The layer `UnderDebugging` changes the color of segments and intersections accessed from the path-printing methods.

While identifying classes, we noticed that two context-dependent use cases, "Solving with right-hand rule" and "Solving with Trémaux," which were not identified as layers, also crosscut multiple classes. The selected algorithm changes not only the behavior of the virtual robot instance but also the enabling configuration for the menu items in GUI components. Thus, we can identify two other layers, i.e., `RightHandRule` and `Tremaux`, that cut across two classes, i.e., `Robot` and `View`. This observation reveals that a context-dependent use case that does not share contexts with other use cases can crosscut multiple classes, and could be a layer. To determine whether such a use case should be identified as a layer, we must assess the use case scenario carefully to determine whether the context-dependent use case includes multiple objects, or to postpone the decision until we design the classes.

After designing the classes, we define the detailed specifications of the contexts. In the previous examples, we provide the specification of each context as

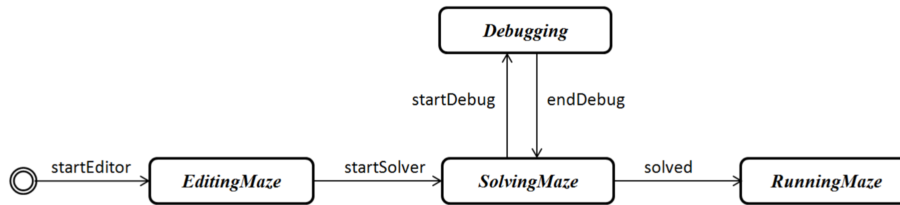


Fig. 7. Context transitions and events in the maze-solving simulator

```
1 global contextgroup MazeUI() {
2   activate EditingMaze from startEditor to startSolver;
3   activate SolvingMaze from startSolver to solved;
4   activate RunningMaze from solved to never;
5   activate Debugging from startDebug to endDebug;
6   context Displaying is in cflow(call(void Simulator.printPath()));
7   activate UnderDebugging when Debugging && when Printing;
8 }
```

Fig. 8. Context group for UI in the maze-solving simulator

a `boolean` type method call implemented using an `if` expression in ServalCJ. In this case study, we follow a different approach. There are apparent state changes in the simulator user interface. First, the interface provides the maze editor to the user. Then, it provides the menus and tools to solve the maze. During maze solving, the user can switch to the debugging mode. After solving the maze, the user interface provides menus and tools to run the optimized path. Each state corresponds to each global context in Table 6, and, by observing use case scenarios, we explicitly identify events by following the method described in [31]. The contexts and events are summarized in Fig. 7.

Using events, we specify each global context (Table 6) as follows.

```
EditingMaze:: after startEditor until startSolver.
SolvingMaze:: after startSolver until solved.
RunningMaze:: after solved.
Debugging:: after startDebug until endDebug.
```

Each condition is read as a term in temporal logic that holds after the specified event (e.g., `startEditor`) after another specified event (e.g., `startSolver`). ServalCJ provides *active until expressions* that correspond to such temporal logic terms (Fig. 8).

How to determine whether we should use explicit or implicit events will be addressed in future work. One possible criterion is performance. Implicit events impose additional overhead on the application (because each conditional in `if` is evaluated before each call of a partial method); therefore, if events are explicit in the specification, we should consider using explicit events. Another criterion

is modularity. Explicit events easily raise the scattering problem or the fragile pointcut problem [37], and implicit events are the solutions to these problems. For the global contexts of the maze-solving simulator, we apply explicit events because events are apparent in the specification, and the specification is unlikely to change.

The context "displaying" in Table 6 is identified as cflow, which means that the subject of this context is a particular control flow. For this purpose, ServalCJ provides the `cflow` construct that declares that the specified context holds under the specified control flow. In Fig. 8, this context is declared to be active under the `printPath` call control flow on an instance of `Simulator` (line 6).

The contexts "rightHand" and "tremaux" are specified as a `boolean` type method call, similar to the cases in the conference guide system and CJEdit.

```
activate RightHandRule if(sim.isRightHandRule());
activate Tremaux if(sim.isTremaux());
```

7 Discussion on Modularity

The case studies demonstrate our COSE methodology and effectively answer the research questions described in Section 3. In this section, we summarize and validate our results.

7.1 Summary of Results: RQ1

In Section 2.2, we identified several difficulties encountered when developing context-aware applications. Thus, we formed RQ1, "How should contexts and behavior depending on the contexts be elicited from the requirements?." To answer RQ1, we suggested Principles 1 and 2 and analyzed their validity through case studies. The results of the case studies are summarized as follows.

Identification of Contexts and Requirements Variability. As illustrated in Section 4.1, COSE systematically identifies contexts by observing the behavior of the system-to-be, such as use cases and prototypes. Furthermore, we clarify a criterion that should hold for each context, i.e., a context should not be a subcase of other contexts. Requirements variability based on contexts is also represented by context-dependent use cases.

Different Levels of Abstraction. As discussed in Sections 4.1 and 4.4, COSE provides a concretization process for contexts. A context may be composed of other contexts that are less abstract than the composed context. Each level of abstraction of contexts in the specification is also directly represented by the implementation language using composite layers.

Multiple Dependencies Between Contexts and Behavior. As discussed above, given composite layers, layer activation can be triggered by complex activation conditions.

Requirements Volatility in Context Specification. Each context-dependent use case is represented in terms of abstract contexts; therefore, it is robust against changes in detailed specifications of concrete contexts. For example, in the conference guide system, the specification of the outdoor context may change according to the evolution of sensor technologies. Context-dependent use cases that depend on the outdoor context will not be affected by such changes because the detailed specification of the outdoor context is abstracted from the context-dependent use cases. We may also separately perform such changes because the definitions of contexts are encapsulated in context groups in ServalCJ.

Crosscutting of Contexts in Multiple Use Cases. COSE groups a number of behavior variations that are executable under the same contexts and scattered across multiple use cases into a single layer. As discussed in Section 4.2, COSE also provides a guideline to determine when to use COP.

Crosscutting of Behavior Changes. Dynamic changes of contexts and dependent behavior scattered across the whole execution of the program are separated as specifications of contexts and implemented directly using context groups. Specifically, definitions of such changes are specified declaratively and completely separated from the base program.

Interferences Between Behaviors. The case studies show that there are dependencies between layers (e.g., outdoors and indoors are exclusive variations of behavior), and COSE clarifies such dependencies due to the orthogonality and exclusiveness of the variables used in the context conditions. These conditions are straightforwardly implemented using composite layers in ServalCJ, and the dependencies are ensured by the implementation language.

Modular Translation to the Implementation. The layers and classes identified in Sections 4.2 and 4.3 are implemented directly in layers and classes in ServalCJ. Context specifications (Section 4.4) are directly implemented in context groups in ServalCJ. Each requirement in the specification is not scattered across multiple modules in the implementation, and each module is not entangled with multiple requirements.

In summary, the case studies reveal that the factors that change system behavior are "candidates" for contexts, and each context can be represented as a Boolean variable. A criterion to identify contexts can be derived from this representation of contexts, i.e., each context at the abstract level should not depend on other contexts. A context and other contexts should be orthogonal or, if they are not orthogonal, they should be exclusive. This criterion enhances the exhaustiveness of contexts and makes it easy to discuss the equivalence between contexts.

7.2 Summary of Results: RQ2

The answer to RQ2, "When should we apply COP rather than other development methods?," is represented by Principles 3 and 4, and to validate that, we must

further discuss the validity of the decision made in the case studies because there are other alternatives to implement such variations.

We can validate it using Tables 4 and 5. First, the layers `HasSchedule` and `HasNetwork` crosscut multiple classes; thus, the same concern may scatter over those classes if we naively implement them using `if` statements. Applying design patterns may also produce this scattering problem. Extracting such scattered code as a common superclass requires an additional class hierarchy, which may be orthogonal to the existing hierarchies. Applying multiple inheritance, mixins [11], and traits [48] makes it difficult to look at all classes that are composed of the same context-dependent behavior. In contrast, layers in COP provide a good solution to encapsulate such concerns. More importantly, techniques other than COP make it difficult to separate behavior changes from the base program, which is possible in (some variants of) COP languages.

In contrast, the `Outdoors`, `Indoors` and `StaticMap` layers in Table 5 exist in only the `Map` class; thus, they do not appear to contribute to the separation of crosscutting concerns. However, from Table 4, we observe that `Outdoors` consists of two use cases implemented by different methods. Therefore, using `if` statements would result in scattering of the same conditions over those methods. We can avoid this scattering by, for example, allowing the `Map` object to have a state of the current situation and by defining behavioral variations for each state using the state design pattern. The problem with applying design patterns is the scattering and tangling of behavioral changes. The state changes of the `Map` object are triggered by external environment changes, which are observed by the framework. We must embed state changes of the `Map` object by implementing appropriate event handlers of possibly multiple modules (e.g., the Wifi and GPS related classes). Thus, it is difficult to localize the overall state changes in the `Map` object. By applying COSE with appropriate COP languages, we can separate such context changes into a single module.

Similar discussion holds in the case study of CJEdit. However, in the maze-solving robot simulator example, the case wherein the context-dependent use cases whose contexts are not shared with other use cases are also identified as layers. This indicates that, while the principles hold, there are cases where we should postpone the decision to implement variations of context-dependent behavior using layers until we are designing classes.

7.3 Summary of Results: RQ3

The implementation in ServalCJ discussed in Section 5 implies that the implementation is *directly obtained* from the requirements in our approach. There are injective mappings from layers and contexts discovered in the requirements to those in the implementation language. Thus, this mapping promotes separation of concerns in that requirements are not scattered across several modules in the implementation, and each module is not entangled with a number of requirements.

The implementations in the case studies rely on the specific linguistic constructs provided by ServalCJ. To answer RQ3 "How do COP mechanisms sup-

Table 9. Comparison with other activation mechanisms

	ContextJ	**AOP+COP**	**EventCJ**	**ServalCJ**
Separation of context-dependent behavior[15]	a	a	a	a
Separation of context changes	n/a	a	a	a
Expressing relations between layers and contexts	n/a	n/a	a	a
Implicit activation	n/a	n/a	n/a	a

port predictable control of changes in context-dependent behavior?," we identify the properties that the implementation languages should have to make COSE effective, and we compare ServalCJ with other languages and implementation techniques, such as ContextJ [7], EventCJ [29, 30], and a pseudo AOP language with a dynamic layer activation mechanism (similar to the one discussed in Section 2 of [29]), with respect to those properties. Table 9 summarizes the comparison. The leftmost column shows the numbers and titles of the following sections.

We do not argue that programming languages that do not support the features listed below are not useful in COSE. In such languages, we can still apply useful workarounds to implement specifications organized by COSE, which would not be a poor choice in some circumstances, such as the availability of libraries and a development environment, and programmer preference. Nevertheless, Table 9 indicates that recent progress in COP languages effectively supports COSE, which will be good input for future language design.

Separation of Context-dependent Behavior. First, in COSE, the implementing language should separate context-dependent behavior that is dynamically enabled and disabled from the base program. The layers of COP languages provide an effective way to achieve this. Each partial method implements the context-specific behavior of the base method, and a layer packs all partial methods executable under the same context into a single module. Besides COP, other programming paradigms, such as AOP and feature-oriented programming (FOP) [42], also provide such modularization mechanisms; however, for these paradigms, we require an additional mechanism for dynamic composition of modules. For example, dynamic aspect deployment [10] may be applied for this purpose.

Separation of Context Changes. We can also see that, in COSE, specifications and implementations of dynamic changes of contexts and dependent

[15] ServalCJ (and EventCJ) only supports the layer-in-class style. Thus, the same layer may be scattered across multiple classes. In fact, such layers exist in both case studies. This scattering can be addressed by supporting the class-in-layer style in the syntax.

behavior are also separated from other specifications and modules, respectively. From an implementation perspective, such dynamic changes can be easily scattered over the whole application execution. Such scattering behavior can be avoided using the pointcut-advice mechanism in AspectJ [36] (provided that it is also equipped with some imperial layer activation mechanism) or other COP languages with AOP features, such as EventCJ and JCop [9].

In some COP languages, layer activation is controlled in a *per-thread* manner whereby the generation of the event activating the layer and layer activation occur synchronously. In such languages, it is difficult to separate dynamic behavior changes. For example, in ContextJ, layer activation is expressed using `with`-blocks, which ensures that layers are active only within the explicitly specified dynamic scope.

```
with (activeLayers) { onResume(); }
```

However, context changes are triggered by external events that asynchronously occur with the dynamic behavior change. For example, in this case, we must remember the active layer within the body of the event handler that handles the change of contexts to activate context-dependent behavior that does not appear in the scope of the event handler:

```
void someEventHandler(Event e) {
  activeLayers.add(Outdoors);
}
```

In this case, the scattering problem is readily encountered, and the base program is entangled with the concerns about dynamic changes of behavior.

Expressing Relations Between Layers and Contexts. From COSE, we also see that a behavior variation may depend on multiple contexts. For example, from Table 4, we see that the use case "Using a static map," which is implemented in the layer `StaticMap`, depends on both the outdoors and indoors contexts, one of which, i.e., outdoors, is further decomposed into two contexts, i.e., `WifiAvailable` and `GPSAvailable`. To separate context-dependent behavior from the detailed specification of contexts, such an abstraction mechanism is necessary. From an implementation perspective, composite layers [30], which are supported by EventCJ and ServalCJ, are useful for this purpose.

Implicit Activation. In most existing COP languages, we must specify the join point where the context change occurs explicitly. In COP languages with AOP features, we perform such specification using the pointcut sublanguage. In COP languages with `with`-blocks, we explicitly inject the layer activation block into the base program. However, from the case studies, we have learned that a more declarative way to specify the condition that activates the corresponding context is used heavily in the context specification, which is directly implemented using the implicit layer activation mechanism provided by ServalCJ (i.e., the `if` condition that specifies the condition when the corresponding context is active).

This indicates that, even though it currently suffers performance problems, the implicit layer activation mechanism can be a strong tool to implement dynamic behavior changes modularly from the specification.

It is also possible to translate implicit layer activation manually into an explicit activation by identifying the join points where the condition is changed. However, with multiple join points, we must list all of them, which is an error-prone task. Furthermore, explicitly specifying join points using a pointcut often raises the fragile pointcut problem [37].

7.4 Open Issues

Our preliminary case studies on COSE raise the following open issues that should be explored.

First, all case studies in this paper are simple. Although these case studies demonstrate the effectiveness of COSE, they do not guarantee success in more complex cases. In large systems, we may have a large number of dynamic behavior changes, some of which would be context dependent. Eliciting contexts from such systems may be time consuming. Furthermore, in all case studies, the target system is standalone and implemented using a single programming language. We should not assume that the results of the case studies imply that we can easily apply COSE to distributed systems implemented using multiple programming languages.

Second, COSE represents variations of context-dependent behavior using use cases. There may be some cases in which we prefer to use methods other than use cases, such as feature diagrams and goal models. The results in this paper do not guarantee that we can apply similar context-oriented extensions to such methods.

Third, the case studies do not convey compelling results regarding the costs and benefits of COSE. The results ensure modularity of the products. However, they do not reveal how such modularity affects the real software production process and the quality of its products. We believe that COSE would have a significant impact on software development, in particular on software maintenance, because it provides comprehensive abstractions, clarifies complex relations between contexts and behavior, and provides good modularity. However, this should be validated through a number of control experiments. Furthermore, the principles explained in Section 3.2 should be validated through a number of demonstration experiments and industrial software development.

Finally, as mentioned above, there are open performance issues with implicit activation, which is heavily used in the case studies. The performance problem is not significant in the case studies; however, this assumption will not always hold in larger applications. In some cases, we may optimize implicit activation, but such optimization may not be feasible in other cases. The case studies do not provide a concrete criterion for when implicit activation is preferable (because, e.g., it enhances modularity) or when other mechanisms, such as event-based activation, should be used (e.g. due to the performance considerations).

8 Future Research Roadmap

In this paper, we have presented COSE and proposed that it can be employed for the effective development of context-aware applications. Specifications systematized by COSE effectively represent different levels of abstraction of contexts, which makes the system robust with respect to changes in the detailed definitions of contexts. Context-dependent use cases are used to discover a layer, i.e., a modularization unit in COP, from the specifications. The injective mapping from specifications to implementations ensures that each specification in the requirements is not scattered across multiple modules in the implementation, and each module is not entangled with multiple requirements. The comparison among several implementation techniques shown in Section 7.3 reveals the key linguistic constructs that make COSE effective and indicates important research directions for context-oriented software development.

This paper has presented preliminary studies on COSE. Although these studies reveal that our approach is promising, there are a number of open issues. In this section, we discuss our future research roadmap.

8.1 Systematizing Context Identification

The applications mentioned in the case studies are simple, and the number of identified contexts is not large. In large systems, the number of "candidates for contexts" will be very large. Furthermore, the system-to-be will be described using natural languages including diagrams in inconsistent syntax. In some cases, such descriptions will be scattered over various resources, such as text documents, spreadsheets, and emails. This unstructured piling up of descriptions can easily results in a situation whereby conceptually equal contexts are described in different words and notations.

In Section 4, we listed the factors that change the system behavior as candidates for contexts. This is the most fundamental property of contexts. To identify contexts systematically and deal with a large number of candidates for contexts, more precise criteria to find candidates for contexts are required. For example, for a factor that changes the system behavior to be identified as a context in COP, it should affect the behavior of a number of objects in the system. Moreover, all contexts in the case studies are external with respect to the affected entities.

From this perspective, we plan to develop a systematic context elicitation process that is applicable in the early stages of requirements elicitation. Some work in requirements engineering, such as context-dependent domain analysis [19], will be a good starting point.

8.2 Requirements Based on Other Methods

Using use cases is a very effective way to identify the functional requirements of the system-to-be. Use cases do not require special languages to describe them;

thus, people from various backgrounds can understand them easily. Nevertheless, they effectively describe system behavior. Furthermore, they prevent hasty design; design methods based on use cases have been well studied.

However, using use cases is not a panacea. For example, they are not suitable for representing non-functional requirements, which are better specified declaratively elsewhere, or for describing the requirements specifications of platforms, such as operating systems and frameworks. There are also a number of methods for analyzing requirements that are not based on use cases. It is natural to ask whether it is possible to apply methods similar to that described in this paper to other requirements analysis methods.

Goal-oriented methods for requirements engineering [18], [40] are complementary approaches suitable for eliciting requirements variability and constraints. Non-functional requirements are derived from their *soft goals*. Their variability and constraints may depend on executing contexts. Although a goal-based approach for contextualization has been proposed previously [3], further research should be conducted to explore, for example, approaches to align goal-based approaches and use-case-driven approaches.

Feature modeling presents a compact representation of all products of a software product line. Feature models are represented by means of feature diagrams [34]. Features provide requirements for architectures (including non-functional requirements) and reusable functions. At the programming language level, layers in COP resemble features in FOP [5], [50]. This similarity indicates that we may develop a context-oriented extension of FOSD [4]. For example, some existing work in this field [14] would be a good starting point.

Application of the context-oriented software development described in this paper to these major requirements engineering methods is a future challenge.

8.3 Evaluation

To ensure that our methodology is effective, it is necessary to perform further evaluation. For example, we must evaluate the costs and benefits of our methodology, and the validity of the decision to use layers to implement context-dependent behavior rather than other mechanisms through controlled experiments that compare our methodology with other software development methods. It is difficult to conduct controlled experiments, and derivation of quantitative evaluations would be a length process. Meanwhile, we think that it is important to conduct a number of demonstration experiments to collect experience by applying our approach. In particular, we believe that the application of our methodology to industrial software development is particularly important.

Since one purpose of our study has been to enhance modularity, an evaluation will be performed from this perspective. For example, an experimental study of how our approach makes it easy to deal with volatile requirements regarding contexts and analysis of the effects of requirement changes should be performed.

8.4 Implicit Activation

In two of the three case studies, all contexts are implemented by means of layer activation triggered by conditionals (i.e., `if` expressions in ServalCJ). As mentioned above, this implies the importance of implicit layer activation. However, there is a performance problem with implicit layer activation. A naive implementation strategy is to evaluate the condition that specifies when the corresponding context is active at every call of the layered methods, and when that condition holds and the corresponding context is not active, then that context is activated. This strategy will not produce a serious problem if the number of layered method calls is not so high. However, in the case where calls of layered methods frequently and repeatedly occur (e.g., where calls of layered methods are included within a loop statement), this strategy may result in serious performance problems.

Thus, developing an optimization mechanism for implicit layer activation so that the evaluation of the context condition occurs only when necessary is an important research topic. There are several approaches for this purpose.

One approach is to develop an ad hoc method that optimizes parts of the program where calls of layered methods may occur frequently, such as loop statements. For example, if we can determine that the context condition will never change during the execution of the loop, we can rewrite the loop so that the context condition is evaluated just once at the entrance of that loop.

For a more effective approach, we may explore a method to statically analyze when the value of the context condition changes. For example, assuming that c is a condition for the context `C`, if we can derive a pair of predicates (p, q) for which it can easily be checked that $(p \wedge p \implies c) \wedge (q \wedge q \implies \neg c)$, then we can insert evaluations of c where the values for p or q change. We are currently considering an application of predicate abstraction for model checking for this purpose.

Although it is desirable to limit changes of a given context condition in a small amount of code, it is generally possible that the change of context condition can occur anywhere in the program execution, which requires a whole program analysis. To make the whole program analysis lightweight and feasible in the case when the whole code is not available for analysis, it is also necessary to study the application of whole program analysis without the whole program [2] for COP programs.

The emphasis on implicit activation does not mean that event-based activation of contexts is not necessary. First, event-based activation should be used where layered methods are frequently called and optimization of implicit layer activation is difficult for some reason. There are also cases where the specification of a context is defined in terms of events (even though this did not occur in our case studies). For example, there may be a specification of stateless objects whose contexts are changed by clicking buttons. In this case, it is better to implement context activation in an event-based manner than to introduce a state for each object to manage context activation using the implicit activation mechanism. There are also cases where context changes can be observed from both conditions and events.

The problem is that there are no clear guidelines for when to use implicit layer activation and when to use the event-based mechanism. To create such guidelines, we must study this problem from both programming language and programming practice perspectives. From the programming language perspective, as mentioned above, it is necessary to determine the feasibility of efficient implementation of implicit activation. Meanwhile, formalization of implicit activation is also desirable to precisely study the semantics of implicit activation. We think that implicit activation (of layers) is a special case of functional reactive programming (FRP) [20] in that the change of the condition (value) reactively changes the result of the activation (computation). FRP is considered a special case of implicit activation (of behavior) by viewing it as a way to propagate values in a constraint graph of variables and expressions. It is possible that both have some shared foundations in continuous constraint solving. Understanding implicit activation in terms of FRP may further clarify the semantics of implicit activation.

From the programming practice perspective, through a number of other case studies, we plan to discover common *patterns* in context activation, which will serve as guidelines.

8.5 Distributed, Multi-language Environment

Both case studies in this paper are standalone applications written in a single programming language. However, in real products, systems are implemented using multiple programming languages and sometimes comprise a number of components and services over networks. There are two problems with applying our methodology to such systems.

First, to the best of our knowledge, ServalCJ is the only language that has all the desirable properties shown in Section 7.3. We must explore how to realize the mechanism supported by ServalCJ in a wide range of programming languages including those suitable for high performance computing such as C and C++, and scripting languages such as JavaScript.

Second, little COP research has been devoted to sharing the same context among multiple application processes. Sharing a context among processes over a network is possible in programming languages that support network-transparent communications between processes such as ContextErlang [47]. Further research is required to support network-transparent contexts in other programming models and develop a mechanism to share contexts among multiple programming languages, which may communicate with each other over the network.

Based on these technical elements, we will study the applicability of COSE to more realistic and sophisticated software development situations.

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. Karim Ali and Ondřej Lhoták. Whole-program analysis without the whole program. In *ECOOP'13*, volume 7920 of *LNCS*, pages 378–400, 2013.
3. Raian Ali, Fabiano Dalpiaz, and Paolo Giorgini. Goal-based self-contextualization. In *CAiSE 2009*, pages 37–43, 2009.
4. Sven Apel and Christian Kästner. On overview of feature-oriented software development. *Journal of Object Technology*, 8(5):49–84, 2009.
5. Sven Apel, Thomas Leich, Marko Rosenmüller, and Gunter Saake. FeatureC++: On the symbiosis of feature-oriented and aspect-oriented programming. In *GPCE'05*, pages 125–140, 2005.
6. Malte Appeltauer, Robert Hirschfeld, Michael Haupt, Jens Lincke, and Michael Perscheid. A comparison of context-oriented programming languages. In *COP'09*, pages 1–6, 2009.
7. Malte Appeltauer, Robert Hirschfeld, Michael Haupt, and Hidehiko Masuhara. ContextJ: Context-oriented programming with Java. *Computer Software*, 28(1):272–292, 2011.
8. Malte Appeltauer, Robert Hirschfeld, and Hidehiko Masuhara. Improving the development of context-dependent Java application with ContextJ. In *COP'09*, 2009.
9. 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.
10. Ivia Aracic, Vaidas Gasiunas, Mira Mezini, and Klaus Ostermann. An overview of CaesarJ. In *Transactions on Aspect-Oriented Software Development I*, volume 3880 of *LNCS*, pages 135–173, 2006.
11. G. Bracha and W. Cook. Mixin-based inheritance. In *OOPSLA 1990*, pages 303–311, 1990.
12. 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.
13. Nicolás Cardozo, Sebastián González, Kim Mens, Ragnhild Van Der Straeten, Jorge Vallejos, and Theo D'Hondt. Semantics for consistent activation in context-oriented systems. *Information and Software Technology*, 58:71–94, 2015.
14. Nicolás Cardozo, Folfgang De Meuter, Kim Mens, and Sebastián González. Features on demand. In *VaMoS'14*, 2014.
15. Stefano Ceri, Florian Daniel, Federico M. Facca, and Maristella Matera. Model-driven engineering of active contet-awareness. *World Wide Web*, 10:387–413, 2007.
16. Pascal Costanza and Theo D'Hondt. Feature description for context-oriented programming. In *DSPL'08*, 2008.
17. 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.
18. Anne Dardenne, Axel van Lamsweerde, and Stephen Fickas. Goal-directed requirements acquisition. *Science of Computer Programming*, 20:3–50, 1993.
19. Brecht Desmet, Jorge Vallejos, Pascal Costanza, Wolfgang De Meuter, and Theo D'Hondt. Context-oriented domain analysis. In *CONTEXT 2007*, volume 4635 of *LNAI*, pages 178–191, 2007.
20. Conal Elliott and Paul Hudak. Functional reactive animation. In *ICFP'97*, pages 263–273, 1997.

21. Sebastián González, Micolás Cardozo, Kim Mens, Alfredo Cádiz, Jean-Christophe Libbrecht, and Julien Goffaux. Subjective-C: Bringing context to mobile platform programming. In *SLE'11*, volume 6563 of *LNCS*, pages 246–265, 2011.
22. Sebastián González, Kim Mens, and Alfredo Cádiz. Context-oriented programming with the ambient object systems. *Journal of Universal Computer Science*, 14(20):3307–3332, 2008.
23. Sebastián González, Kim Mens, Marius Colãcious, and Walter Cazzola. Context traits: Dynamic behaviour adaptation through run-time trait recomposition. In *AOSD'13*, pages 209–220, 2013.
24. Karen Henrichsen and Jadwiga Indulska. A software engineering framework for context-aware pervasive computing. In *PERCOM'04*, 2004.
25. 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.
26. Robert Hirschfeld, Pascal Costanza, and Oscar Nierstrasz. Context-oriented programming. *Journal of Object Technology*, 7(3):125–151, 2008.
27. Ivar Jacobson, Magnus Christerson, Patrik Jonsson, and Gunnar Övergaard. *Object-Oriented Software Engineering: A Use Case Driven Approach*. Pearson Education, 1992.
28. Ivar Jacobson and Pan-Wei Ng. *Aspect-Oriented Software Development with Use Cases*. Pearson Education, 2005.
29. 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.
30. Tetsuo Kamina, Tomoyuki Aotani, and Hidehiko Masuhara. Introducing composite layers in EventCJ. *IPSJ Transactions on Programming*, 6(1):1–8, 2013.
31. Tetsuo Kamina, Tomoyuki Aotani, and Hidehiko Masuhara. Mapping context-dependent requirements to event-based context-oriented programs for modularity. In *Workshop on Reactivity, Events and Modularity (REM 2013)*, 2013.
32. Tetsuo Kamina, Tomoyuki Aotani, and Hidehiko Masuhara. Generalized layer activation mechanism through contexts and subscribers. In *MODULARITY'15*, pages 14–28, 2015.
33. Tetsuo Kamina, Tomoyuki Aotani, Hidehiko Masuhara, and Tetsuo Tamai. Context-oriented software engineering: A modularity vision. In *MODULARITY'14*, pages 85–98, 2014.
34. 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.
35. Roger Keays and Andry Rakotonirainy. Context-oriented programming. In *MobiDE'03*, pages 9–16, 2003.
36. 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.
37. Christian Koppen and Maximilian Störzer. PCDiff: Attacking the fragile pointcut problem. In *European Interactive Workshop on Aspects in Software*, 2004.
38. Alexiei Lapouchnian and John Mylopoulous. Modeling domain variability in requirements engineering with contexts. In *ER 2009*, volume 5829 of *LNCS*, pages 115–130, 2009.
39. 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.

40. Lin Liu and Eric Yu. Designing information systems in social context: a goal and scenario modelling approach. *Information Systems*, 29(2):187–203, 2004.
41. David Lorge Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, 1972.
42. Christian Prehofer. Feature-oriented programming: A fresh look at objects. In *ECOOP'97*, volume 1241 of *LNCS*, pages 419–443, 1997.
43. 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.
44. 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.
45. Mohammed Salifu, Bashar Nuseibeh, Lucia Rapanotti, and Thein Than Tun. Using problem descriptions to represent variability for context-aware applications. In *VaMoS 2007*, 2007.
46. Mohammed Salifu, Yujun Yu, and Bashar Nuseibeh. Specifying monitoring and switching problems in context. In *RE'07*, pages 211–220, 2007.
47. Guido Salvaneschi, Carlo Ghezzi, and Matteo Pradella. ContextErlang: A language for distributed context-aware self-adaptive applications. *Science of Computer Programming*, 102(1):20–43, 2014.
48. Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew P. Black. Traits: Composable units of behaviour. In *ECOOP 2003*, volume 2743 of *LNCS*, pages 248–274, 2003.
49. 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.
50. Fuminobu Takeyama and Shigeru Chiba. Implementing feature interactions with generic feature modules. In *Software Composition 2013*, volume 8088 of *LNCS*, pages 81–96, 2013.
51. Jorge Vallejos, Sebastián González, Pascal Costanza, Wolfgang De Meuter, Theo D'Hondt, and Kim Mens. Predicated generic functions: Enabling context-dependent method dispatch. In *SC'10*, volume 6144 of *LNCS*, pages 66–81, 2010.