Towards Type-Safe JCop

A type system for layer inheritance and first-class layers

Hiroaki Inoue
Grad. School of Informatics
Kyoto University, Japan
hinoue@kuis.kyoto-u.ac.jp

Atsushi Igarashi
Grad. School of Informatics
Kyoto University, Japan
igarashi@kuis.kyoto-u.ac.jp

Malte Appeltauer
SAP Innovation Center,
Potsdam, Germany
malte.appeltauer@sap.com

Robert Hirschfeld
Hasso-Plattner-Institut
Univ. of Potsdam, Germany
hirschfeld@hpi.uni-potsdam.de

ABSTRACT

This paper describes a type system for JCop, which is an extension of Java with language mechanisms for context-oriented programming. A simple type system as in Java, however, is not sufficient to prevent errors due to the absence of invoked methods because interfaces of objects can change at run time by dynamic layer composition, a characteristic feature of context-oriented programming. Igarashi, Hirschfeld, and Masuhara have studied a type system for dynamic layer composition, but their type system is not directly applicable to JCop due to JCop-specific features such as layer inheritance, first-class layers, and declarative layer composition. We discuss how their type system can be extended to these language features.

Categories and Subject Descriptors

D.1.5 [**Programming Techniques**]: Object-Oriented Programming; D.3.3 [**Programming Languages**]: Language Constructs and Features

General Terms

Languages

Keywords

Context-oriented programming, first-class layers, layer inheritance, type systems

1. INTRODUCTION

Context-Oriented Programming (COP) [5] is an approach to describing context-dependent behavioral variations in a program modularly. The main language constructs for COP are *layers* and *dynamic layer activation*. A layer is a collection of *partial methods*, which modify the behavior of objects by running before, after, or around the objects' methods. Dynamic layer activation refers to an ability to make partial methods in a layer effective (or ineffective) during program execution. Since one layer can contain partial methods for different objects (or classes), dynamic layer activation can change the behavior of multiple objects at once. So, a layer can modularize behavioral changes that cross-cuts over several object or class definitions. Different COP languages provide different constructs for activation (and deactivation).

JCop [1] is an extension of Java with language constructs for COP. It introduces not only basic COP mechanisms described above but also advanced ones, such as inheritance of layer implementations, first-class layers, subtyping between layer types, and declarative layer composition to enhance flexibility. Typechecking in the current JCop compiler, however, is not sufficient to prevent errors due to the absence of invoked methods at run time; it is because the compiler performs typechecking only after translating a given JCop program to plain Java classes.

Typechecking JCop programs is more difficult than Java programs because layer (de)activation can dynamically change the interfaces of objects (if a layer adds new method to base classes). Igarashi, Hirschfeld and Masuhara [7] have studied this problem and developed a formal calculus ContextFJ, which models a core of COP, with a provably sound type system. ContextFJ, however, supports only the basic COP mechanisms, namely, global and second-class layers and block-style layer activation and it is not obvious how their type system can extend to other features such as first-class layers, layer inheritance, and declarative layer composition of JCop.

In this paper, we extend the type system of ContextFJ to main additional features of JCop. In particular, we focus on first-class layers, inheritance of layers, and subtyping between layer types and how they affect typing. We also discuss typing for layer *de*activation, which has been known to be difficult to deal with.

The rest of the paper is organized as follows. We review main features of JCop and state the problem in Section 2.

COP '14 July 28 - August 01 2014, Uppsala, Sweden

ACM 978-1-4503-2861-6/14/07/ ...$15.00.
http://dx.doi.org/10.1145/2637066.2637073.

Section 3 is the main part of this paper and informally describes our type system for JCop. We discuss other JCop features, limitations, and possible improvements with related and future work in Section 4 and conclude in Section 5.

2. JCOP

In this section, we review the main features of the JCop language [1] along with the introduction of terminology and then describe the problem we will address. As a running example, we consider programming a graphical computer game called *RetroAdventure*, inspired by [2]. In this game, the user-controlled hero character wanders around the world. Here are (skeletons of) the class definitions of the hero, which has the method to `move` to a given `dir`ection by a certain distance, and the world in which the hero lives.

```
public class Hero {
  Position pos;
  public void move(Direction dir){
    pos = ...;
     /* changes pos according to dir */
  }
}
public class World {
  ...
}
```

2.1 Layers and Partial Methods

Now, let's introduce the notion of weather, which affects how the hero can move, into the game. For example, in a rainy weather the hero gets slow and, in a stormy weather, the hero cannot move to the direction to which he would like to move. We will define layers to represent these weather conditions to affect the behavior of the method `move` given above. Here are the definitions of these layers:

```
public layer Rainy {
  /* partial method */
  public void Hero.move(Direction dir){
    pos = ...;
    /* the distance of move is smaller */
  }
}
public layer Stormy {
  /* partial method */
  public Hero.move(Direction dir){
    proceed(randomDirection(dir));
  }
  /* layer-local method */
  Direction randomDirection(Direction dir){
    return ...;
      /* add randomness to dir */
  }
}
public layer Sunny { ... }
```

In JCop, a layer is defined just like a Java class. A layer includes *partial method* declarations, which resemble ordinary methods except that the name is in a qualified form such as `Hero.move`, which designates which method to modify. We often call original method definitions *base methods* and the class that includes the base method *base class*. In layer `Rainy`, the partial method calculates the hero's new position differently from the original, while, in layer `Stormy`, the partial method uses `proceed`, which is used to invoke the original behavior (with a different argument). A partial method can be given without a corresponding base method, in which case a layer introduces new behavior to the base class and `proceed` should not be used. The method named `randomDirection` (without qualification) is a layer-local method, which is considered a member of a layer and hence cannot use `proceed`, either.

2.2 Layer Activation

JCop provides a few different mechanisms to *activate* a layer to make partial methods effective. In this subsection, we describe only the so-called block-style activation, which activates a layer for a certain duration of execution. The code below shows how a rainy weather is realized:

```
// in main method
Rainy rainy = new Rainy();
with(rainy){
  /* method dispatch will be influenced by
      Rainy */
}
```

In JCop, a layer has to be instantiated by the keyword `new` before it is activated, just as an object in Java is instantiated by using a class definition. The `with` statement above activates a layer instance `rainy` and executes the following block. Inside the block, every invocation of `move` on the hero will execute the partial method in `Rainy`. When the partial method body has `proceed` calls (in this case it does not), the base implementation is called.[1] Layer activation has dynamic extent in the sense that, when a method is invoked inside a `with` block, the execution of the method body is also affected by the layer instance activated by the callee. JCop also provides the `without` statement, which deactivates the given layer instance, and the `withoutall` statement, which deactivates all the instances of the given layer.

Note that layer instances are first-class citizens in JCop and the name of a layer works as a type for its instances. It is possible to store a layer instance in a variable or a field of an object, pass as a method argument, and even define its own state by declaring layer-local fields.

2.3 Layer Inheritance

In JCop, a layer can inherit definitions from another layer by using the keyword `extends`, just as Java classes. If the weather layers have many definitions in common, it is a good idea to define a superlayer `Weather` and concrete weather layers as its sublayers. In the example below, the layers add two methods to `World`: `getWorldText` to generate some formatted message to tell the current status of the world and `getWeatherInfo` to return the current weather. The former is common among all the weathers but the latter depends on the concrete weather.

```
public layer Weather{
  public String World.getWorldText(){
    ..... + this.getWeatherInfo() + ....;
  }
  public String World.getWeatherInfo(){
    return "not implemented";
  }
}
public layer Rainy extends Weather{
  public String World.getWeatherInfo(){
    return "rain";
```

[1] To be more precise, `proceed` calls the next partial method, when more than one layer instance that has a partial method for the same base method is activated. In such a case, the definition in a more recently activated layer instance is called first.

```
  }
}
public layer Stormy extends Weather{
  public String World.getWeatherInfo(){
    return "storm";
  }
}
```

(For simplicity, `getWeatherInfo` in `Weather` returns a bogus string. In JCop, one may define `Weather` as an abstract layer and leave the implementation of `getWeatherInfo` abstract.)

Just as in Java, the `extends` clauses define the subtyping relation between (layer) types. For example, a variable of type `Weather` can store instances of `Rainy` or `Stormy`, and be used for activation:

```
Weather weather;
weather = someCondition() ? new Rainy()
                          : new Stormy();
with(weather){...}
```

Note that method dispatch is based on the dynamic type of activated layer instances and the static type `Weather` is not relevant. If first-class layer instances were not supported, we would have to use conditional branches to choose what layer to activate. Such programming is possible but more cumbersome. In JCop, layer `Layer` is declared implicitly as the superlayer of all layers just as `Object` in Java. This layer implements some layer-specific functionality, and is often used as the type of a variable to store layers.

2.4 Declarative Layer Composition

In addition to the block-style (de)activation mechanisms, JCop provides *declarative layer composition* [2]—a mechanism to (de)activate layer instances by using AspectJ-like *pointcuts* [12] and *context classes*. An example of a context class is given below.

```
public layer DebugLayer{
  public void Hero.move(Direction dir){
    print(dir);
    proceed(dir);
  }
}
contextclass DebugMode{
  DebugLayer debug = new DebugLayer();
  boolean isDebug = false;
  public void setDebug(boolean b){isDebug = b;}

  when(isDebug) &&
  on(public String Hero.move(Direction))
    : with(debug);
}
```

The context class `DebugMode` has two fields and one method, followed by a pointcut with advice. The pointcut (the part before the colon) matches a join point representing the execution of `move` in `Hero` under the condition that `isDebug` is true; the advice (the part after the colon) specifies that the layer instance `debug` should be activated. As a result, while an instance of `DebugMode` is activated and `isDebug` is true,[2] every invocation of `move` in `Hero` first prints out the given direction.

2.5 Problem

[2]We omit how an instance of a context class can be activated, as it is not very important for the present paper; see Appeltauer and Hirschfeld [2] for details.

As mentioned in Secion 1, typechecking JCop programs is more difficult than Java programs due to the fact that a layer can add new methods to base classes and availability of a method depends on layer activation at run time. For example, the following layer `Logging` is supposed to record the weather information when `getWeatherInfo` is invoked on the world, but, since `getWeatherInfo` does not exist in the original implementation of `World`, the success of `proceed` (or, in general, the invocation of a partial method that does not have a corresponding base definition) depends on whether a weather layer has been activated or not.

```
public layer Logging{
  public String World.getWeatherInfo(){
    return logging(proceed());
  }
}
...
with(new Logging()){
  new World().getWeatherInfo();
  // proceed() may cause an error.
}
```

In fact, the current JCop compiler accepts the `with` statement above and a run-time exception may be thrown during the execution.

Igarashi, Hirschfeld, and Masuhara [7] have studied this problem and developed a type system for a formal calculus ContextFJ, a COP extension of Featherweight Java [8]. Although the type system is proven sound in the sense that a well typed program does not cause errors due to the invocation of non-existent (partial) methods or dangling `proceed`, the target language, which only supports second-class layers and `with`, is too simple to be applied to JCop.

In this paper, we extend the type system of ContextFJ to deal with the JCop features described above.

3. TYPE SYSTEM FOR JCOP

In this section, we describe our type system for JCop; we start with reviewing the main ideas behind the type system for ContextFJ, as it is the basis of our work, and then discuss how we deal with subtyping between layer types, layer deactivation by `without`, and declarative layer composition.

3.1 Type System for ContextFJ

The main ideas behind the type system for ContextFJ is (1) the introduction of the *requires* relation between layers and (2) the approximation of activated layer instances at each program point.

The *requires* relation states that, if layer L_1 requires L_2, then L_2 has to be activated before L_1 is activated. For example, `Logging` requires `Weather` because `proceed` inside `Logging` succeeds only when the method of the same name (namely `getWeatherInfo`) is made available in `World` by activating an instance from one of the weather layers. Following ContextFJ, a programmer is supposed to declare the requires relation by using a `requires` clause as follows:

```
public layer Logging requires Weather {...}
```

In general, one layer can require more than one layer by listing layer names.

The type system also estimates an (under-)approximation of the set of layers activated at each program point and uses it to check whether a certain method exists or not at a given program point. For example, the type checker allows

`getWeatherInfo` of `World` to be invoked only when one of the weather layers is in the estimated active layer set. Thanks to declared `requires`, it is easy to compute such under-approximation: at the beginning of a partial method in layer L, which `requires` $\{L_1, \ldots, L_n\}$, the approximated layers are $\{L_1, \ldots, L_n\}$; inside a `with` statement that activates an instance of L', L' is added to the approximation. In case L' requires other layers, it is checked that the layers that L' requires are included by the approximated layers outside the `with` statement. For example, to activate an instance of `Logging`, the `with` statement has to be surrounded by another `with` to activate a weather layer instance or appear in a partial method of the layer that requires `Weather`.

Remark. Igarashi, Hirschfeld, and Masuhara considered a layer activation mechanism called `ensure`, which is slightly different from `with`, but, as was shown by Inoue [9], it turns out that the difference does not really matter and the same idea works also for `with`.

3.2 Making Layer Subtyping Safe

As we already discussed, the `extends` relation between layers defines the subtyping between layer types and an instance of a sublayer can be substituted for that of a superlayer. The substitutivity applies to the `requires` relation, too: when a layer requires another layer, any instance of a sublayer of the required layer can be substituted for the superlayer. For example, although `Logging requires Weather`, it will not cause any problem to activate an instance of `Rainy` before the activation of a `Logging` instance.

```
with(new Rainy()){
  with(new Logging()){
    ...
  }
}
```

In fact, there is subtle interaction among `requires` clauses, inheritance of partial methods, and subtyping. Consider layer `MoreLogging`, which is defined to be a sublayer of `Logging` and requires another layer `OtherInfo`.

```
public layer OtherInfo{...}
public layer MoreLogging extends Logging
             requires OtherInfo{...}
...
Logging log =
  someCondition() ? new MoreLogging()
                  : new Logging();
with(new Rainy()){
  with(log){/*???*/}
}
```

At first, the activation of `log` seems fine according to the typing rule for activation described in the last subsection because its static type `Logging` requires only `Weather` and an instance of `Rainy` has been activated already. However, the activated instance may be that of `MoreLogging`, which also requires `OtherInfo`, which is *not* activated! So, if the block body invokes a partial method in `MoreLogging`, which may call partial methods that exist only in `OtherInfo`, the execution results in an error. So, a sublayer cannot require more layers than its parent, or a layer has to require at least all the layers that its sublayers require. This restriction is natural since a layer is kind of a function that takes a set of classes as input and a function type is not covariant in argument types.

One may expect it is fine to allow a sublayer to require *fewer* layers than its parent so that an instance of the sublayer can be activated in more occasions. Obviously, it is not safe, either (even without first-class layers). For example, consider the layer definition below

```
public layer AnotherLogging extends Logging
    requires none {}
```

(Here, `none` is used to denote explicitly that this layer requires no other layers; it can be omitted.) Accepting this definition amounts to creating a copy of `Logging`, which had to require `Weather`. In general, a sublayer may inherit partial methods from its parent and they may require layers specified in the `requires` from the parent. One way to deal with a sublayer requiring fewer layers is to *recheck* inherited partial method definitions under the new `requires` clause, which we would like to avoid.

So, our current design is that a sublayer automatically inherits the `requires` clause as it is. One exception is that, if a sublayer overrides all partial methods in the parent, the sublayer can specify fewer layers in the `requires` clause (because there is no need to recheck).[3]

Layer type hierarchy. The discussion above seems to imply that `Layer`, which resides at the top of the layer hierarchy, has to require all layers in a program. However, `with(l){...}` where `l`'s static type is `Layer` is then virtually impossible because we cannot activate all the layers in advance (especially in an open-world setting). This situation can be problematic because `Layer` is often used in practice as a type to denote any layer instance, just as `Object` in Java is used to denote any object. For example, one may want to store different layer instances in an array of type `Layer[]` and activate (some of the) stored layer instances. To address this problem, we introduce a special layer `BaseLayer`, which requires no layers, to the layer hierarchy. Our observation here is that such use of `Layer` is valid if layers do not require any other layers. We explain the role of `Layer` and `BaseLayer` below:

- `Layer` is put at the top of the layer hierarchy and works as a supertype of all layer types. It is prohibited to activate a layer instance whose static type is `Layer`, but it is allowed to store a layer instance to a variable of type `Layer` or perform a downcast to a specific layer type. If a layer definition does not specify its superlayer and the `requires` clause is not empty, then the layer is implicitly assumed to extend `Layer`.

- `BaseLayer` is a direct sublayer of `Layer`. Its `requires` clause is considered empty, so an instance of any layer that extends `BaseLayer` can be safely activated anywhere. If a layer definition does not specify its superlayer and the `requires` clause is empty, then the layer is implicitly assumed to extend `BaseLayer`.

`BaseLayer` works as a top type for layers that can be activated anytime, so the code like below is possible:

```
BaseLayer[] layers = new BaseLayer[10];
layers[0] = new Rain();
layers[1] = new OtherInfo();
with(layers[i]){ ... }
```

[3]More precisely, the overriding method should not call `superproceed`, which calls a partial method of the same name defined in a superlayer.

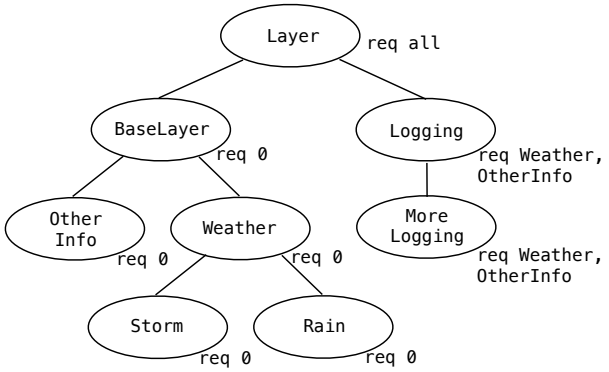

Figure 1: Layer type hierarchy.

Figure 1 illustrates how the layers we have seen so far are put in the layer hierarchy. (Here, `req` stands for `requires` and $\emptyset$ the empty set.)

3.3 Safe Layer Swapping

It is not easy to guarantee that layer *de*activation does not lead to an error due to the absence of partial methods. In fact, layer deactivation is almost neglected in previous work. In the first type system for ContextFJ [6], a layer cannot introduce a new method and so the interface of an object does not change for its lifetime, making deactivation trivially safe. In the second type system for ContextFJ by Igarashi, Hirschfeld, and Masuhara [7], a layer can introduce a new method but deactivation is dropped for the sake of simplicity. In the study of a variant of the second type system [9], the first author of the present paper considered deactivation and set the rule as follows: "`without` is allowed only if the target layer is not required by any other layers". This condition certainly makes deactivation safe but it is also prohibitively restrictive.

In this paper, instead of dealing with deactivation directly, we try to support one important idiom that uses deactivation, that is, *layer swapping*. Suppose we want to switch the weather of the world to stormy, then we would write something like

```
withoutall(Weather){
  with(new Stormy()){...}
}
```

but this code does not satisfy the condition mentioned above because `Weather` is required by another layer.

To treat such layer swapping, we introduce new mechanisms: `swappable` layer declaration and `swap` statements. The `swappable` layer declaration means that its sublayers can be swapped by a `swap` statement explained below but no layers can require its sublayers (the swappable layer itself can be required by another layer). A `swap` statement is written as follows.

`swap(`*activation layer*, *deactivation type*`) { ...}`

The *activation layer* is an expression whose static type must be a subtype of *deactivation type*, which in turn has to be swappable. It deactivates all instances of *deactivation type* (and its sublayers), and activates the *activation layer*.

For example, the weather changing code can be written as follows:

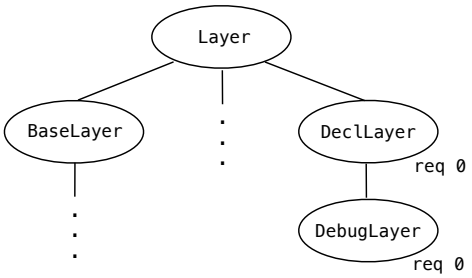

Figure 2: Refined layer type hierarchy with `DeclLayer`.

```
public swappable layer Weather{...}
...
Weather w = new Stormy();
with(w){
  swap(new Rainy(),Weather){
    // deactivate Stormy and activate Rainy
  }
}
```

Typechecking a `swap` statement is easy: the *deactivation type* and all its subtypes are removed from the estimated active layer set and the static type of the *activation layer* is added. The reason we still require that no sublayer of a swappable layer be required by other layers is that, for example, if there is a layer directly requiring `Stormy`, replacing it with an instance of `Rainy` may not be safe.

3.4 Safe Declarative Layer Composition

It is even more difficult to make declarative layer composition safe. This is because layer (de)activation by pointcut in a context class happens independently of the control flow that the type system can see.

So, we have decided to have a separate layer family for layers used in context classes. More specifically, we introduce `declarative` layers, which can be specified by putting `declarative` before `layer` in a layer definition. For example, the example code of Sec 2.4 becomes

```
public declarative layer DebugLayer{...}
```

Declarative layers obey the following rules:

- Only `declarative` layers can appear in advice in context classes and they cannot be used in a block-style layer activation.
- A new layer `DeclLayer` is introduced as a sublayer of `Layer` and the superlayer of all declarative layers. (See Figure 2.)
- A declarative layer can neither require any other layers nor be required by any other layers so that it can be activated or deactivated at any moment.

3.5 Implementation

We have implemented a part of the type system described so far, more concretely, most of what is described in Section 3.2, in JCop. The JCop compiler translates JCop code into Java code and, before the translation, it performs simple name checks. We added typechecking after this phase.

4. DISCUSSION

In this section, we will discuss about other JCop-specific features, related work, and future work.

Other JCop Features. JCop supports `before` and `after` modifiers for partial method declarations to execute partial methods before and after the base method, respectively. Fortunately, they will not cause additional problems. One can specify certain layers and context classes to be `staticactive`, that is, always active during the execution. So, the type system can take advantage of its presence, for example, by adding staticactive layers to the estimated layer set. The reflection mechanism of JCop offers operations to change currently activated layers at run time. Therefore, static type checking becomes invalid. JCop offers some layer local methods that link (de)activation of a layer: an `onWith` method is called when the enclosing layer is activated. This feature is motivated by layer composition from layer dependencies, and similar to problems treated in [11] and [4]. However, this functionality is not safe because it uses reflection extensively. Dealing with event-based layer composition is left for future work.

Related Work. We refer readers to Igarashi, Hirschfeld, and Masuhara [7] for basic comparisons with other composition mechanisms and type systems for COP. We focus on dynamic deactivation, which is one of the new problems, here.

Swappable layers in Section 3.3 resemble atomic layers in ContextL [4], in which mutual exclusion between layers can be specified and activation of an atomic layer may automatically deactivates another layer in conflict. Actually it seems a little verbose to specify the swappable layer name such as `Weather` in our `swap` statement.

Recent work by Kamina, Aotani, and Igarashi [10] proposes another way to deal with type-safe deactivation. In their proposal, a layer can specify other layers that must have been activated before the present layer is activated just as `requires` in our setting. A significant difference is that, at every method invocation, those required layers will be *automatically activated* by the run-time system. This automatic activation seems to be convenient in many cases but we do not think it is always possible to specify such layers, especially in the presence of subtyping. For example, `Logging` requires *one of the weather layers* but it may not be easy to choose one layer.

Context Petri Nets [3], a context-oriented extension of Petri Nets, support various kinds of declarations of dependency between layers, such as implication, requirement, and exclusion. Exclusion can express, for example, that at most one weather layer can be activated. It is interesting future work to support such declarations to help typechecking (rather than introducing `swap`).

Future Work. The description of the type system is only informal. One important piece of future work would be to formalize the type system and prove its soundness.

Specifying a concrete layer *implementation* by `requires` is not always convenient. We expect it increases flexibility of the type system by introducing *interfaces* of layers in the same spirit as Java interfaces.

We have not fully investigated the interaction between our type system with other features in Java, such as concurrency, generics, and lambda, although we expect most of them are orthogonal.

Last not but least, we should examine how good our type system is in practice through writing applications. For example, it is interesting to see whether the invariant rule for `requires` (see Section 3.2) is not too restrictive. To do so, we should implement our type system first.

5. SUMMARY

In this paper, we have introduced a type system for JCop. For this purpose, we have extended the type system for ContextFJ by layer subtyping and first-class layer objects. We also have discussed about layer deactivation mechanism and other JCop-specific features. We are going to finish implementing the ideas described here, and make it publicly available.

Acknowledgments. We appreciate valuable comments and suggestions form the anonymous reviewers.

6. REFERENCES

[1] M. Appeltauer and R. Hirschfeld. The JCop language specification. Technical report, HPI, University of Potsdam, 2012.

[2] M. Appeltauer, R. Hirschfeld, and J. Lincke. Declarative layer composition with the JCop programming language. *Journal of Object Technology*, 12, 2013.

[3] N. Cardozo, S. González, K. Mens, R. Van Der Straeten, and T. D'Hondt. Modeling and analyzing self-adaptive systems with context petri nets. In *Proc. of TASE*, pages 191–198. IEEE, 2013.

[4] P. Costanza and T. D'Hondt. Feature descriptions for context-oriented programming. In *SPLC (2)*, pages 9–14, 2008.

[5] R. Hirschfeld, P. Costanza, and O. Nierstrasz. Context-oriented programming. *Journal of Object Technology*, 2008.

[6] R. Hirschfeld, A. Igarashi, and H. Masuhara. ContextFJ: A minimal core calculus for context-oriented programming. In *Proc. of the FOAL2011*, pages 19–23, Mar. 2011.

[7] A. Igarashi, R. Hirschfeld, and H. Masuhara. A type system for dynamic layer composition. In *Proc. of FOOL*, Oct. 2012.

[8] A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM TOPLAS*, 2001.

[9] H. Inoue. A proof of soundness of type system for dynamic layer composition. Undergraduate honors thesis, Kyoto University, 2013. In Japanese.

[10] T. Kamina, T. Aotani, and A. Igarashi. On-demand layer activation for type-safe deactivation. In *Proc. of COP'14*, Uppsala, Sweden, July 2014.

[11] T. Kamina, T. Aotani, and H. Masuhara. EventCJ: a context-oriented programming language with declarative event-based context transition. In *Proc. of ACM AOSD*, pages 253–264. ACM, 2011.

[12] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An Overview of AspectJ. In *Proc. of ECOOP'01*, volume 2072 of *Springer LNCS*, pages 327–353, 2001.