FOAL 2010 Proceedings

Proceedings of the Ninth Workshop on Foundations of Aspect-Oriented Languages

held at the
Ninth International Conference on
Aspect-Oriented Software Development

March 15, Rennes, France

Gary T. Leavens, Shmuel Katz, and Mira Mezini (editors)

CS-TR-10-04
March 2010

School of Electrical Engineering and Computer Science
4000 Central Florida Blvd.
University of Central Florida
Orlando, Florida 32816-2362, USA

Contents

Preface

Aspect-oriented programming is a paradigm in software engineering and programming languages that promises better support for separation of concerns. The Ninth Foundations of Aspect-Oriented Languages (FOAL) workshop was held at the Ninth International Conference on Aspect-Oriented Software Development in Rennes, France, on March 15, 2010. This workshop was designed to be a forum for research in formal foundations of aspect-oriented programming languages. The call for papers announced the areas of interest for FOAL as including: semantics of aspect-oriented languages, specification and verification for such languages, type systems, static analysis, theory of testing, theory of aspect composition, and theory of aspect translation (compilation) and rewriting. The call for papers welcomed all theoretical and foundational studies of foundations of aspect-oriented languages.

FOAL logos courtesy of Luca Cardelli

The goals of this FOAL workshop were to:

- Make progress on the foundations of aspect-oriented programming languages.
- Exchange ideas about semantics and formal methods for aspect-oriented programming languages.
- Foster interest within the programming language theory and types communities in aspect-oriented programming languages.
- Foster interest within the formal methods community in aspect-oriented programming and the problems of reasoning about aspect-oriented programs.

The workshop was organized by Shmuel Katz (Technion–Israel Institute of Technology, Israel), Gary T. Leavens (University of Central Florida, USA), and Mira Mezini (Darmstadt University of Technology, Germany). We are very grateful to the program committee, which was chaired very ably by Klaus Ostermann.

We thank the organizers of AOSD 2010 for hosting the workshop.

Message from the Program Committee Chair

The ninth FOAL workshop continues to be one of the primary forums for foundational work on aspect-oriented software development. As in the past, each paper was subjected to full review by at least three reviewers. I am grateful to the program committee members for their dedication, insightful comments, attention to detail, and the service they provided to the community and the individual authors.

The members of the program committee were:

- Klaus Ostermann (Program Committee Chair, Philipps-Universität Marburg, Germany)
- Sven Apel (University of Passau),
- Eric Bodden (T.U. Darmstadt),
- Erik Ernst (University of Aarhus),
- David Lorenz (The Open University of Israel),
- Hidehiko Masuhara (University of Tokyo),
- Hridesh Rajan (Iowa State University),
- James Riely (DePaul University),
- Eric Tanter (University of Chile),
- Elena Zucca (University of Genoa)

The sub-reviewers, whom I also thank, were: Eugenio Moggi and Marko Rosenmueller.

I am also grateful to the authors of submitted works. Ten papers were submitted for review this year. Of these, the program committee selected nine for presentation at the workshop and publication in the proceedings, but one paper was later withdrawn, as the authors could not attend the workshop.

The program was rounded out with a discussion section.

Finally, I would like to thank the other members of the organizing committee of FOAL—Shmuel Katz, Gary T. Leavens, and Mira Mezini— for their work in guiding us toward another inspiring workshop.

Klaus Ostermann
FOAL '10 Program Committee Chair
Philipps-Universität Marburg, Germany

StrongRelaxAJ: integrating adaptability of RelaxAJ and expressiveness of StrongAspectJ

Tomoyuki Aotani
Graduate School of Arts and Sciences
University of Tokyo
aotani@graco.c.u-tokyo.ac.jp

Manabu Toyama
Graduate School of Arts and Sciences
University of Tokyo
touyama@graco.c.u-tokyo.ac.jp

Hidehiko Masuhara
Graduate School of Arts and Sciences
University of Tokyo
masuhara@acm.org

ABSTRACT

A sketch of StrongRelaxAJ is presented. StrongRelaxAJ is an extension to AspectJ with a type system for around advice that integrates the ones in RelaxAJ and StrongAspectJ. In other words, StrongRelaxAJ employs the type-relaxed weaving mechanism in RelaxAJ for better adaptability of around advice, and supports type variables and explicit signatures of proceed for better expressiveness without relying on dangerous and annoying dynamic casting on the return values from `proceed`.

Categories and Subject Descriptors

D.3.3 [**PROGRAMMING LANGUAGES**]: Language Constructs and Features—*Polymorphism*

General Terms

Design, Languages

Keywords

Aspect-Oriented Programming, Around Advice, Type-Relaxed Weaving, AspectJ, RelaxAJ, StrongAspectJ, StrongRelaxAJ

1. INTRODUCTION

Around advice is one of the unique and powerful features of the pointcut and advice mechanism. It allows programmers not only to replace the operations with others without directly modifying the source code but also to change parameters and return values of operations by using `proceed`.

Defining a good type system for around advice is one of the challenges in statically typed aspect-oriented programming (AOP) languages that employ the pointcut and advice mechanism. Because a type system conservatively accepts "safe" programs, it constrains the adaptability of around advice.

Recent studies revealed and solved problems of type-safety and expressiveness in AspectJ [3, 5], which is one of the widely used statically typed AOP languages.

RelaxAJ [6], which is an extension to AspectJ, improves the adaptability of AspectJ's around advice by relaxing the restriction on its return type. While a piece of around advice and its *target join point* on which the advice is executed must have the same return type for type safety in AspectJ, this is not required any more in RelaxAJ. Instead, it guarantees type safety by ensuring that the return values of around advice are safely used within the program.

StrongAspectJ [2] is another extension to AspectJ, which supports type-safe generic around advice. A piece of around advice in StrongAspectJ is safely evaluated on each target join point. Intuitively, it is achieved by ensuring that a piece of around advice always returns the return values of `proceed`.

This position paper points out the problems of expressiveness in RelaxAJ as well as the problems of adaptability in StrongAspectJ, and proposes *StrongRelaxAJ* as our solution. StrongRelaxAJ integrates the adaptability of RelaxAJ and genericity of StrongAspectJ. In other words, it solves the problems of expressiveness in RelaxAJ as well as solves the problems of adaptability in StrongAspectJ. In the position paper, we roughly explain its syntax and type checking rules by using a concrete example. Its formalization and implementation are left for future work.

The rest of the paper is organized as follows. Section 2 gives a brief overview of RelaxAJ and StrongAspectJ. Section 3 presents examples that cannot be achieved by either RelaxAJ or StrongAspectJ, and Section 4 shows a sketch of StrongRelaxAJ. After discussing related work in Section 5, Section 6 concludes the position paper and lists our future work.

2. BACKGROUND: RELAXAJ AND STRONGASPECTJ

This section presents brief overviews of RelaxAJ and StrongAspectJ along with a code fragment that implements a popup window.

2.1 Base code: creating a popup window

Suppose we have an image editor in which one can manipulate images by applying various filters (e.g., Gaussian blur filters) and see a preview of the filter's effect in a popup window. Listing 1 shows the method `showPreview` that creates a popup window for previewing.

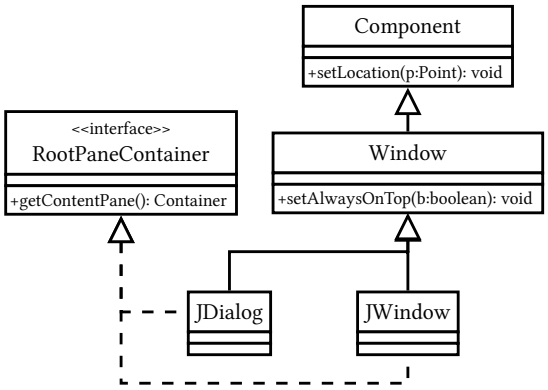


Figure 1: Relationship between `Component`, `Window`, `JDialog`, `JWindow` and `RootPaneContainer`

```
1 void showPreview(JFrame mainWin, MyImage image){
2   JWindow popup = new JWindow(mainWin);
3   MyCanvas canvas = new MyCanvas(image);
4   JButton closeButton = new JButton("close");
5   popup.getContentPane().add(canvas);
6   popup.getContentPane().add(closeButton);
7 }
```

Listing 1: Popup window for previewing

It takes two parameters, namely `mainWin`, which is the main window of the application, and `image`, which is the processed image the user will see. `MyCanvas`, which is a subclass of `JPanel`, draws the processed image. The popup window `popup`, which is an instance of `JWindow`, contains a canvas and button. If the button is clicked then the popup window is closed.

2.2 RelaxAJ

RelaxAJ is an extension to AspectJ, which has a novel type-checking rule for the return types of around advice. A piece of around advice in RelaxAJ can have a return type that is not a subtype of the target join points' on which the advice is executed. In AspectJ, the return type instead must be a subtype of the target join points'.

Of course the return type cannot be any type in RelaxAJ. It must be consistent with the types as which the return values are used in the program.

Assume we want to create a modal dialog instead of a simple popup window in the above example so as not to leave previews outdated. One of the easiest ways to achieve it is to use `JDialog` with the modal flag, instead of `JWindow`.

Listing 2 shows a piece of around advice that implements

```
1 RootPaneContainer around(Frame frame)
2   :call(JWindow.new(Frame))&&args(frame){
3   return new JDialog(frame, true);
4 }
```

Listing 2: Around advice that replaces JWindow with JDialog

```
1 <T extends Component>
2 T around(Frame frame)
3   :call((JDalog||JWindow).new(Frame))&&args(frame)
4   :T proceed(Frame){
5   T popup = proceed(frame);
6   popup.setLocation(DEFAULT_LOC);
7   return popup;
8 }
```

Listing 3: Around advice that specifies the location of the popup window

the idea. It simply creates a new modal `JDialog` object and returns it when a `JWindow` object is to be created.

Note that the return type, which is `RootPaneContainer`, is a supertype of the target join point's return type, which is `JWindow`. It is not valid in AspectJ because it requires the return type is a subtype of the return types of its target join points.

RelaxAJ accepts the advice when it is applied only to the line 2 in Listing 1 because its return value is used only as `RootPaneContainer` within the program and thus the replacement is safe.

2.3 StrongAspectJ

StrongAspectJ is another extension to AspectJ, which supports type-safe generic around advice. The genericity and type-safety are achieved by using (bounded) type variables to declare the return types of around advice and also `proceed`. Although AspectJ implicitly decides the return type of `proceed`, StrongAspectJ does not. It is given by the programmer through a dual advice signature.

Assume a popup window is an instance of either `JDialog` or `JWindow`, and we want to specify the location where the window appears. This can be achieved by calling `setLocation` that is defined in `Component`.

Listing 3 is a piece of the around advice in StrongAspectJ that catches the popup window object and calls `setLocation` on it. Line 1 declares the type variable `T` whose upper bound is `Component`. It is used as the return type of the advice. Line 4 is the dual advice signature that declares the return and argument types of `proceed`: here its return type is `T` and its argument type is `Frame`.

If the base program is type safe, the woven program is also type safe. This is because (1) the return type of each target join points (`JDialog` or `JWindow`) is always a subtype of `Component` (see Figure 1) so that no type error occurs within the advice, and (2) the return types of the advice and `proceed` are always the same so that the values returned by the advice can be used safely as the original values.

3. EXAMPLES THAT NEED AN INTEGRATED LANGUAGE

By integrating RelaxAJ and StrongAspectJ, we can implement more adaptive and interesting aspects. This section presents two examples that cannot be achieved in either RelaxAJ or StrongAspectJ alone but can be achieved in the integrated language.

3.1 Specifying return type of proceed in type-relaxing advice

```
1 RootPaneContainer around(Frame frame)
2   :call(JWindow.new(Frame))&&args(frame){
3   if(POPUP_MODAL) return new JDialog(frame, true);
4   else{
5     JWindow popup=(JWindow)proceed(frame);
6     JOptionPane.showMessageDialog(popup,ALERT);
7     return popup;
8   }
9 }
```

Listing 4: Using dynamic casting to use return values of `proceed` as a `JWindow`

Suppose that we want to make the popup window modal only if `POPUP_MODAL` is true. Otherwise, we show a message dialog that warns danger of out-of-date previews along with the original popup window. It can be achieved in RelaxAJ by defining a piece of around advice shown in Listing 4

The problem here is the use of a cast operator at line 5. It is necessary because RelaxAJ simply adapts AspectJ's typing rule for `proceed`.

The return type of `proceed` is the same to the one of the around advice, that is, `RootPaneContainer`. On the other hand, to set `popup` as the parent window of the message dialog (`JOptionPane`) through `showMessageDialog`[1], its static type must be a subtype of `Component`, which is incompatible with `RootPaneContainer`.

We should be able to omit dynamic casting because it is obvious that `proceed` always returns a `JWindow` object. One way to achieve it is to add StrongAspectJ's dual advice signature and the typing rules to RelaxAJ. StrongAspectJ allows programmers to declare the return type of `proceed`. `JWindow` is a valid return type here because it is (1) a supertype of the return type of the target join points (`JWindow` itself) and also (2) a subtype of the return type of the advice (`RootPaneContainer`).

Of course StrongAspectJ does not accept such advice because its return type is invalid: it must be a subtype of the return types of the target join points in StrongAspectJ, but here `RootPaneContainer` is a supertype, not a subtype, of `JWindow`.

3.2 Abstracting the return type of around advice by using type variables

RelaxAJ provides no way to write a piece of around advice whose return value of is used as two or more types incompatible with each other. In other words, the return type of a piece of around advice must be one type in RelaxAJ.

It is natural to control orders of windows in GUI programs. Listing 5 extends the base program (Listing 1) so that the popup window stays above all other windows. Line 7 is added where `popup` is used as a `Window` because `setAlwaysOnTop`, which is an instance method defined in `Window`, is called on it.

The around advice declaration in Listings 2 and 4 cannot be compiled with the above extended program. This is because the return type cannot be relaxed to `RootPaneContainer`. As mentioned before, the return value is used as not only a `RootPaneContainer` but also a `Window`.

Modifying the return type is not a solution because there is no such a type that is a subtype of `RootPaneContainer` and `Window` and a supertype of `JDialog` and `JWindow`.

[1] `showMessage(Component,Object)` is a static method in `JOptionPane`

```
1 void showPreview(JFrame mainWin, MyImage image){
2   JWindow popup = new JWindow(mainWin);
3   MyCanvas canvas = new MyCanvas(image);
4   JButton closeButton = new JButton("close");
5   popup.getContentPane().add(canvas);
6   popup.getContentPane().add(closeButton);
7   popup.setAlwaysOnTop(true);
8 }
```

Listing 5: Create a popup window that stays above all other windows

```
1 <T extends RootPaneContainer & Window>
2 T around(Frame frame)
3   :call(JWindow.new(Frame))&&args(frame)
4   :JWindow proceed(Frame){
5   if(POPUP_MODAL) return new JDialog(frame, true);
6   else{
7     JWindow popup=proceed(frame);
8     JOptionPane.showMessageDialog(popup,ALERT);
9     return popup;
10  }
11 }
```

Listing 6: Around advice in StrongRelaxAJ with an explicit signature of proceed and type variable

4. STRONGRELAXAJ

We propose StrongRelaxAJ, which is a hybrid of RelaxAJ and StrongAspectJ. StrongRelaxAJ has two additional language features, namely *explicit signature of proceed* and *type variables* to the type-relaxed weaving mechanism in RelaxAJ. An explicit signature of proceed helps us to omit dynamic casts shown in Section 3.1. Type variables are used to define a piece of around advice whose return values are used as two or more incompatible types shown in Section 3.2.

This section first explains how the StrongRelaxAJ around advice looks by showing an example. Then it explains about explicit signature of proceed and type variables.

4.1 Around advice in StrongRelaxAJ

The syntax of around advice in StrongRelaxAJ is similar to StrongAspectJ. A piece of around advice in StrongRelaxAJ has declarations of type variables and the signature of proceed.

Listing 6 is a piece of around advice in StrongRelaxAJ. It is a modified version of the advice in Listing 3.1 that works with the extended base code shown in Listing 5.

Line 1 declares the type variable `T` whose upper bounds are `RootPaneContainer` and `Window`. It is used as the return type of the advice in Line 2 instead of `RootPaneContainer`. Line 4 declares the signature of `proceed`.

Note that we does not use dynamic casting at line 7. Because the return type of `proceed` is `JWindow`, we can use its

return value as a `JWindow` object.

4.2 Explicit signature of proceed

The return type of `proceed` in a piece of around advice must be (1) a supertype of the return types of the target join points and also (2) a supertype of the return types of around advice that may be called by `proceed`. In other words, StrongRelaxAJ does not need any relationships between the return type of a piece of around advice and its `proceed` unlike AspectJ, StrongAspectJ and RelaxAJ. This does not break type-safety because `proceed` never calls the around advice that encloses it.

Let's look at the example in Section 4.1. The explicit signature of proceed on Line 4 in Listing 6 satisfies the condition. The return type `JWindow` is not a type variable, and it is clearly a supertype of `JWindow`, which is the return type of the target join points. Because there is no other pieces of advice, the second condition for non variable return types holds too.

4.3 Type variables

Type variables in StrongRelaxAJ are more expressive than the ones in StrongAspectJ. StrongAspectJ uses type variables to ensure that the return value of `proceed` is the return value of the advice. For instance, if the return type of proceed is `T`, which is a type variable, then the enclosing around advice must be `T`.

In addition to the usage, StrongRelaxAJ uses them to declare that the advice returns a value of some type that satisfies the upper bounds. Listing 6 is an example. It uses the type variable `T` to return `JDialog` and `JWindow`. Because each of them is a subtype of `Window` and `RootPaneContainer`, StrongRelaxAJ's type system accepts the return statements.

Note that the return value is used as only a `Window` and a `RootPaneContainer` within the target program, that is, Listing 5. Therefore, type safety is preserved.

5. RELATED WORK

Adding union types to Java [4] gives another solution for the situation in Section 3.2. If it is allowed to use union types, we can declare the return type of the advice as `JWindow`∨`JDialog` instead of using a type variable as in Listing 6. Then RelaxAJ with union types successfully accepts the advice because `JWindow` and `JDialog` are subtypes of `JWindow`∨`JDialog` and `RootPaneContainer` and `Window` are supertypes of `JWindow`∨`JDialog`.

6. CONCLUSIONS AND FUTURE WORK

The position paper presented a sketch of StrongRelaxAJ, which is a hybrid of RelaxAJ and StrongAspectJ. By using explicit signature of proceed, programmers can omit dynamic casting on the return values of proceed. Type variables are used not only to write generic advice but also to declare the return type of a piece of around advice whose return type cannot be described by using only one type.

Dealing with parameter types of `proceed` is one of our future work as RelaxAJ. Formalization and implementation are also our future work. Formalization could be done by extending Featherweight Java for Relaxation (FJR) [6] and StrongAspectJ [2]. Implementation would be done on top of StrongAJ compiler or the AspectBench compiler (abc) [1].

7. REFERENCES

[1] Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Jennifer Lhoták, Ondrej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. abc: An extensible AspectJ compiler. In *Proceedings of AOSD'05*, pages 87–98, 2005.

[2] Bruno De Fraine, Mario Südholt, and Viviane Jonckers. StrongAspectJ: Flexible and safe pointcut/advice bindings. In *Proceedings of AOSD'08*, pages 60–71, 2008.

[3] Erik Hilsdale and Jim Hugunin. Advice weaving in AspectJ. In *Proceedings of AOSD'04*, pages 26–35, 2004.

[4] Atsushi Igarashi and Hideshi Nagira. Union types for object-oriented programming. In *Proceedings of SAC'06*, pages 1435–1441, 2006.

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

[6] Hidehiko Masuhara, Atsushi Igarashi, and Manabu Toyama. Type relaxed weaving. In *Proceedings of AOSD'10*, 2010. To appear.

Translucid Contracts for Aspect-oriented Interfaces

Mehdi Bagherzadeh
Iowa State University,
Ames, IA, USA
mbagherz@cs.iastate.edu

Hridesh Rajan
Iowa State University,
Ames, IA, USA
hridesh@cs.iastate.edu

Gary T. Leavens
University of Central Florida,
Orlando, FL, USA
leavens@eecs.ucf.edu

ABSTRACT

There is some consensus in the aspect-oriented community that a notion of interface between joinpoints and advice may be necessary for improved modularity of aspect-oriented programs, for modular reasoning, and for overcoming pointcut fragility. Different approaches for adding such interfaces, such as aspect-aware interfaces, pointcut interfaces, crosscutting interfaces, explicit joinpoints, quantified typed events, open modules, and joinpoint types decouple aspects and base code, enhancing modularity. However, existing work has not shown how one can write specifications for such interfaces that will actually allow modular reasoning when aspects and base code evolve independently, and that are capable of specifying control effects, such as when advice does not proceed. The main contribution of this work is a specification technique that allows programmers to write modular specification of such interfaces and that allows one to understand such control effects. We show that such specifications allow typical interaction patterns, and interesting control effects to be understood and enforced. We illustrate our techniques via an extension of Ptolemy, but we also show that our ideas can be applied in a straightforward manner to other notions of joinpoint interfaces, e.g. the crosscutting interfaces.

1. INTRODUCTION

In the past decade, the remarkable visibility and adoption of aspect-orientation [28] in research and industrial settings only confirms our belief that new AOSD techniques provide software engineers with valuable opportunities to separate conceptual concerns in software system to enable their independent development and evolution. This same decade of AOSD research has also witnessed an intense debate surrounding two issues: pointcut fragility and modular reasoning. The debate on pointcut fragility focuses on the use of pattern matching as a quantification mechanism [48, 50], whereas that on modular reasoning focuses on the effect of AO modularization on independent understandability and analyzability of modularized concerns [1, 17, 26, 29]. Although the jury is still out, in the later part of the last decade some consensus has begun to emerge that a notion of interfaces may help address questions of pointcut fragility and modular reasoning [1, 13, 20, 29, 41, 47, 49].

1.1 The Problems and their Importance

Although these proposals differ significantly in their syntactic forms and underlying philosophies, the permeating theme is that they provide some notion of explicit interface that abstracts away the details of the modules that are advised (typically referred to as the "base modules") thus hiding such details from modules that advise them (typically referred to as the "crosscutting modules" or "aspects"). Leaving the comparison and contrast of software engineering properties of these proposals to empirical experts, in this paper we focus on studying the effectiveness of such interfaces towards enabling a design by contract methodology for AOSD [1].

Design by contract methodologies for AOSD have been explored before [49, 55], however, existing work relies on behavioral contracts. Such behavioral contracts specify, for each of the aspect's advice methods, the relationships between its inputs and outputs, and treat the implementation of the aspect as a black box, hiding all the aspect's internal states from base modules and from other aspects. To illustrate, consider the snippets shown in Figure 1 from the canonical drawing editor example with functionality to draw points, lines, and a display updating functionality.

Figure 1 uses a proposal for aspect interfaces[2], promoted by our previous work on the Ptolemy language [41]. In Ptolemy, programmers declare event types that are abstractions over concrete events in the program. Lines 10–16 declare an event type that is an abstraction over program events that cause change in a figure. An event type declaration may declare variables that make some context available. For example, on line 11, the changing figure, named `fe`, is made available. Concrete events of this type are created using **`announce`** expressions as shown on lines 5–7.

A significant advantage of such interfaces is that they provide a syntactic location to specify contracts between the aspect and the base code [49] that is independent of both. Following previous work [49, 55], we have added an example behavioral contract to the interface (event type `Changed`) (lines 12-15). This behavioral contract is written in a form similar to our proposal to make comparisons easier. This contract states that any concrete event announcement must ensure that the context variable `fe` is non null and observers (e.g. the `Update` class on lines 17–26) for this event must not modify `fe`.

The first problem with specifying aspect interfaces using behavioral contracts is that they are insufficient to specify the control effects of advice in full generality. For example, with just the be-

[1]This is not to be confused with DBC *using* AOP, where the advice construct is used to represent contract of a method. Rather we speak of the contract between aspects and the base code.

[2]The choice is more of preference than of necessity. Other proposals are equally suitable for this discussion. The reader is encouraged to consider alternatives discussed in Section 4.

```
1 class Fig { }
2 class Point extends Fig {
3  int x; int y;
4  Fig setX(int x){
5   announce Changed(this){
6    this.x = x; this
7   }
8  }
9 }
```

```
10 Fig event Changed {
11  Fig fe;
12  provides  fe != null
13  requires {
14    fe == old(fe)
15  }
16 }
```

```
17 class Update {
18  Update init(){register(this)}
19  when Changed do update;
20  Display d;
21  Fig update(thunk Fig rest,
22                   Fig fe){
23   d.update(fe);
24   invoke(rest)
25  }
26 }
```

Figure 1: A behavioral contract for aspect interfaces using Ptolemy [41] as the implementation language. See Section 2.1 for syntax.

havioral specification of the event type `Changed`, we cannot determine whether the body of the method `setX` will always set the current `x` coordinate to the argument. Such assertions are important for reasoning, which depends on understanding the effect of composing the aspect modules with the base code [44, 49]. In Figure 1, for example, the behavioral contract for `Changed` doesn't serve to alert us to an (inadvertently) missing **`invoke`** expression from the `Update` code that would skip the evaluation of the expression **`this`**`.x = x` in the method `setX`. In AspectJ terms this would be equivalent to a missing **`proceed`** statement from an **`around`** advice. Ideas from Zhao and Rinard's Pipa language [55], if applied to AO interfaces help to some extent, however, as we discuss in greater detail in Section 5, Pipa's expressiveness beyond simple control flow properties is limited.

The second problem with such behavioral contracts is that they don't help us in effectively reasoning about the effects of aspects on each other. Consider another example concern, say `Logging`, which logs the event `Changed`. For this concern different orders of composition with the `Update` concern in Figure 1 could lead to different results. In Ptolemy ordering between aspects can be specified using **`register`** expressions that activate an aspect. In AspectJ **`declare precedence`** serves the same purpose. In one composition where `Update` runs first followed by `Logging`, the evaluation of `Logging` will be skipped, whereas `Logging` would work in the reverse order of composing these concerns. An aspect developer may not, by just looking at the behavioral contract of the aspect interface, reason about their aspect modules. Rather they must be aware of the effects of all aspects that apply to that aspect interface [1, 16, 17]. Furthermore, if any of these aspect modules changes (i.e., if their effects change), one must reason about every other aspect that applies to the same aspect interface.

Finally, even if programmers don't use formal techniques to reason about their programs, contracts for AO interfaces can serve as the programming guidelines for imposing design rules [49,52]. Behavioral contracts for AO interfaces yield insufficiently specified design rules that leave too much room for interpretation, which may differ significantly from programmer to programmer. This may cause inadvertent inconsistencies in AO program designs and implementations, leading to hard to find errors.

1.2 Contributions to the State-of-the-art

The main contribution of this work is the notion of *translucid contracts* for AO interfaces, which is based on grey box specifications [9–12]. A translucid contract for an AO interface can be thought of as an abstract algorithm describing the behavior of aspects that apply to that AO interface. The algorithm is abstract in the sense that it may suppress many actual implementation details, only specifying their effects using specification statements. This allows the specifier to decide to hide some details, while revealing others. As in the refinement calculus, code satisfies an abstract algorithm specification if the code refines the specification [2,36,37], but we use a restricted form that requires structural similarity, to allow specification of control effects.

To illustrate, consider the translucid contract shown in Figure 2. The classes `Fig` and `Point` in this example are the same as in Figure 1. Contrary to the behavioral contract, internal states of the handler methods that run when the event `Changed` is announced are exposed. In particular, any occurrence of **`invoke`** expression in the handler method must be made explicit in the translucid contract. This in turn allows the developer of the class `Point` that announces the event `Changed` to understand the control effects of the handler methods by just inspecting the specification of `Changed`. For example, from lines 5–6 one may conclude that, irrespective of the concrete handler method, the body for the method `setX` on line 6 of Figure 1 will always be run. Such conclusions allow the client of the `setX` to make more expressive assertions about its control flow without considering every handler method that may potentially run when the event `Changed` is announced.

```
1 Fig event Changed {
2  Fig fe;
3  provides  fe != null
4  requires {
5   preserves fe == old(fe);
6   invoke(next)
7  }
8 }
9 class Update {
10  /* ... the same as before */
11  Fig update(thunk Fig rest, Fig fe){
12   refining preserves fe==old(fe){
13    d.update(fe);
14   } ;
15   invoke(rest)
16  }
17 }
```

Figure 2: A translucid contract for aspect interfaces

Making the **`invoke`** expression explicit also benefits other handlers that may run when the event `Changed` is announced. For example, consider the logging concern discussed earlier. Since the contract of `Changed` describes the control flow effects of the handlers, reasoning about the composition of the handler method for logging and other handler methods becomes possible without knowing about all explicit handler methods that may run when the event `Changed` is announced. In this paper we explicitly focus on the use of translucid contract for describing and understanding control flow effects.

To soundly reap these benefits, the translucid contract for the event type `Changed` must be refined by each conforming handler method [2, 36, 37]. We borrow the idea of structural refinement from JML's model programs [45] and enhance it to support aspect-oriented interfaces, which requires several adaptation that we discuss in the next section. Briefly the handler method `update` on lines 11–16 in Figure 2 refines the contract on lines 5–6 because line 15 matches line 6 and lines 12–14 claim to refine the spec-

ification expression on line 5. In summary, this work makes the following contributions. It presents:

- A specification technique for writing contracts for AO interfaces;
- An analysis of the effectiveness of our contracts using Rinard *et al.*'s work [44] on aspect classification that shows that our technique works well for specifying all classes of aspects (as well as others that Rinard *et al.* do not classify);
- A demonstration that besides the AO interface proposal by the previous work of Rajan and Leavens [41], our technique works quite well for crosscut interfaces [49] and Aldrich's Open Modules [1] and a discussion of the applicability of our technique to Steimann *et al.*'s joinpoint types [47], Hoffman and Eugster's explicit joinpoints [21], and Kiczales and Mezini's aspect-aware interface [29]; and
- A comparison and contrast of our specification and verification approach with related ideas for AO contracts.

2. TRANSLUCID CONTRACTS

In this section, we describe our notion of translucid contracts and present a syntax and refinement rules for checking these contracts. We use our previous work on the Ptolemy language [41] for this discussion.[3] However, as we show in Section 4 our basic ideas are applicable to other aspect-oriented programming models. We first present Ptolemy's programming features and then describe the specification features.

2.1 Program Syntax

Ptolemy is an object-oriented (OO) programming language with support for declaring, announcing, and registering with events much like implicit-invocation (II) languages such as Rapide [34]. The registration in Ptolemy is, however, much more powerful compared to II languages as it allows developers to quantify over all subjects that announce an event without actually naming them. This is similar to aspect-oriented languages such as AspectJ [27]. The formally defined OO subset of Ptolemy shares much in common with MiniMAO$_1$ [15], a variant of Featherweight Java [22] and Classic Java [18]. It has classes, objects, inheritance, and subtyping, but it does not have **super**, interfaces, exception handling, built-in value types, privacy modifiers, or abstract methods.

The syntax of Ptolemy executable programs is shown in Figure 3 and explained below. A Ptolemy program consists of zero or more declarations, and a "main" expression (see Figure 1). Declarations are either class declarations or event type declarations.

Declarations. We do not allow nesting of *decl*s. Each class has a name (c) and names its superclass (d), and may declare finite number of fields (*field**) and methods (*meth**). Field declarations are written with a class name, giving the field's type, followed by a field name. Methods also have a C++ or Java-like syntax, although their body is an expression. A binding declaration associates a set of events, described by an event type (p), to a method [41]. An example is shown in Figure 1, which contains a binding on line 19. This binding declaration tells Ptolemy to run the method `update` whenever events of type `Changed` are executed. Implicit invocation terminology calls such methods *handler methods*.

An event type (**event**) declaration has a return type (c), a name (p), zero or more context variable declarations (*form**), and a translucid contract (*contract*). These context declarations specify the types and names of reflective information exposed by conforming events [41]. An example is given in Figure 1 on lines 10–16. In writing examples of event types, as in Figure 1, we show each formal parameter declaration (*form*) as terminated by a semicolon (**;**). In examples showing the declarations of methods and bindings, we use commas to separate each *form*.

[3]Our descriptions of Ptolemy's syntax and semantics are adapted from our previous work [41].

```
prog    ::= decl* e
decl    ::= class c extends d { field*meth* binding* }
          | t event p { form* contract }
field   ::= t f;
meth    ::= t m (form*) { e } | t m (thunk t var, form*) { e }
form    ::= t var,     where var≠this
binding ::= when p do m
e       ::= n | var | null | new c() | e.m(e*) | e.f | e.f = e
          | if (ep) { e } else { e } | cast c e | form = e; e | e; e
          | while (ep) { e } | register(e) | invoke(e)
          | announce p (e*) { e } | refining spec { e }
ep      ::= n | var | ep.f | ep != null | ep == n | ep < n | ! ep | ep && ep
```

where

n	$\in$	$\mathcal{N}$, the set of numeric, integer literals
c, d	$\in$	$\mathcal{C}$, a set of class names
t	$\in$	$\mathcal{C} \cup \{\textbf{int}\}$, a set of types
p	$\in$	$\mathcal{P}$, a set of event type names
f	$\in$	$\mathcal{F}$, a set of field names
m	$\in$	$\mathcal{M}$, a set of method names
var	$\in$	$\{\textbf{this}\} \cup \mathcal{V}$, $\mathcal{V}$ is a set of variable names

Figure 3: Ptolemy's Syntax [41]. Note the new expression `refining` and contracts in the syntax of event types.

Expressions. The formal definition of Ptolemy is given as an expression language [41]. It includes several standard object-oriented (OO) expressions and also some expressions that are specific to announcing events and registering handlers. The standard OO expressions include object construction (**new** c`()`), variable dereference (*var*, including **this**), field dereference ($e.f$), **null**, cast (**cast** t e), assignment to a field ($e_1.f = e_2$), a definition block (t *var* = e_1**;** e_2), and sequencing (e_1**;** e_2). Their semantics and typing is fairly standard [13, 15, 41] and we encourage the reader to consult [41].

There are also three expressions pertinent to events: **register**, **announce**, and **invoke**. The expression **register**(e) evaluates e to an object o, registers o by putting it into the list of active objects, and returns o. The list of active objects is used in the semantics to track registered objects. Only objects in this list are capable of advising events. For example line 18 of Figure 1 is a method that, when called, will register the method's receiver (**this**). The expression **announce** p $(v_1, \ldots, v_n)$ $\{e\}$ declares the expression e as an event of type p and runs any handler methods of registered objects (i.e., those in the list of active objects) that are applicable to p [41]. The expression **invoke**(e) is similar to AspectJ's **proceed**. It evaluates e, which must denote an event closure, and runs that event closure. This results in running the first handler method in the chain of applicable handlers in the event closure. If there are no remaining handler methods, it runs the original expression from the event. **thunk** t ensures that the value of e is an event closure with t being the return type of event closure and hence the type returned by **invoke**(e).

When called from an event, or from **invoke**, each handler method is called with a registered object as its receiver. The call passes an event closure as the first actual argument to the handler method (named `rest` in Figure 1 line 21). Event closures may not be explicitly constructed in programs, neither can they be stored in

fields. They are only constructed by the semantics and passed to the handler methods.

There is one new program expression: refining. A **refining** expression, of the form **refining** *spec* { *e* }, is used to implement Ptolemy's translucid contracts (see below). It executes the expression *e*, which is supposed to satisfy the contract *spec*.

2.2 Specification Features

The syntax for writing an event type's contract in Ptolemy is shown in Figure 4. In this figure, all nonterminals that are used but not defined are the same as in Figure 3.

contract ::= **provides** *sp* **requires** { *se* }
spec ::= **requires** *sp* **ensures** *sp*
sp ::= *n* | *var* | *sp*.*f* | *sp* != **null** | *sp* == *n*
| *sp* == **old**(*sp*) | ! *sp* | *sp* && *sp*
| *sp* < *n*

se ::= *sp* | **null** | **new** *c*() | *se*.*m*(*se**) | *se*.*f* | *se*.*f* = *se*
| **if** (*sp*) { *se* } **else** { *se* } | **cast** *c se* | *form* = *se*; *se*
| **while** (*sp*) { *se* } | *se*; *se* | *form* = *se*; *se* | *se*; *se*
| **register**(*se*) | **invoke**(*se*) | **announce** *p* (*e**) { *e* }
| **next** | *spec* | **either** { *se* } **or** { *se* }

Figure 4: Syntax for writing translucid contracts

A *contract* is of the form **provides** sp **requires** { se }. Here, sp is a specification predicate as defined in Figure 4 and the body of the contract se is an expression that allows some extra specification-only constructs (such as choice expressions).

As discussed previously, sp is the precondition for event announcement. The specification expression se is the abstract algorithm describing conforming handler methods. If a method runs when an event of type p is announced, then its implementation must refine the contract se of the event type p. For example, in Figure 2 the method `update` (lines 11–16) must refine the contract of the event type `Changed` (lines 5–6).

There are four new expression forms that only appear in contracts: specification expressions, **next** expressions, abstract invoke expressions and choice expressions. A specification expression (*spec*) hides and thus abstracts from a piece of code in a conforming implementation [43,45]. The most general form of specification expression is **requires** sp_1 **ensures** sp_2, where sp_1 is a precondition expression and sp_2 is a postcondition. Such a specification expression hides program details by specifying that a correct implementation contains a **refining** expression whose body expression, when started in a state that satisfies sp_1, will terminate in a state that satisfies sp_2 [43, 45]. In examples we use two sugared forms of specification expression. The expression **preserve** sp is sugar for **requires** sp **ensures** sp and **establish** sp is sugar for **requires** `1` **ensures** sp [43]. Ptolemy uses 0 for "false" and non-zero numbers, such as 1, for "true" in conditionals.

The **next** expression, the **invoke** expression and the choice expression (**either** { se } **or** { se }) are place holders in the specification that express the event closure passed to a handler, the call of an event handler using **invoke** and a conditional expression in a conforming handler method. The choice expression hides and thus abstracts from the concrete condition check in the handler method. For a choice expression **either** { se_1 } **or** { se_2 } a conforming handler method may contain an expression e_1 that refines se_1, or an expression e_2 that refines se_2, or an expression **if** (e_0) { e_1 } **else** { e_2 }, where e_0 is a side-effect free expression, e_1 refines se_1, and e_2 refines se_2.

3. ANALYSIS OF EXPRESSIVENESS

To analyze the expressiveness of translucid contracts, in this section we illustrate their application to specify base-aspect interaction patterns discussed by Rinard *et al.* in a previous work [44]. Rinard *et al.* classify base-aspect interaction patterns into: *direct* and *indirect interference*. Direct interference is concerned about control flow interactions whereas indirect interference refers to data flow properties. Direct interference is concerned about calls to **invoke**, which is the Ptolemy's equivalent of AspectJ's **proceed**. Direct interference is further categorized into 4 classes of: augmentation, narrowing, replacement and combination advices. We use the same classification of base-advice interaction for subject-observer interactions. An example, built upon the drawing editor example, is shown for each category of direct interferences.

3.1 Direct Interference: Augmentation

Informally an augmentation handler is allowed to call **invoke** exactly once. Augmentation handler can be an after or before handler. In after augmentation, after the event body the handlers are always executed. The handler `logit` in observer class `Logging` in Figure 5 is an example of an after augmentation. The classes `Point` and `Fig` are the same as in Figure 1. The requirement is "to log the changes when figures are changed". The handler `logit` causes the event body (line 13) to be run first by calling **invoke** and then logs the change in figure.

```
1  Fig event Changed {
2   Fig fe;
3   provides fe != null
4   requires {
5     invoke(next);
6     preserve fe==old(fe)
7   }
8  }
9  class Logging extends Object {
10  when Changed do logit;
11  Log l; /* ... */
12  Fig logit (thunk Fig rest, Fig fe){
13   invoke(rest);
14   refining preserve fe==old(fe) {
15    l.logChange(fe);fe
16   }
17  }
18 }
```

Figure 5: Specifying after augmentation

The interaction between subject and observer is documented explicitly in the event type specification (`Changed`) shown on lines 3-7. Notice that the **invoke** expressions appears exactly once in the event type contract. Thus the base code developers for classes that announce this event can assume that all handlers advising this event would have exactly one call to **invoke** in their implementation and therefore these handlers would be augmentation handlers. Furthermore, **invoke** is called at the beginning of the contract, requiring event handlers to be run after the event body. This imposes another restriction on the implementation of handlers and conveys to the base code developers that not only the handlers are augmentation handler, but also that they will be run after the event body.

Structural similarity is one criterion to be met by handler implementation to refine event specification. In this example structural similarity mandates the handler implementation to have exactly one call to **invoke** at its beginning. This ensures that all handlers advising the event type `Changed` are of type after augmentation.

3.2 Direct Interference: Narrowing

A narrowing handler calls **invoke** at most once, which implies existence of a conditional statement guarding the calls to **invoke**. To illustrate consider the listing in Figure 6, which shows an example of narrowing handler for the drawing editor example. This handler implements an additional requirement for the editor that "some figures are fixed and thus they may not be changed or moved". To implement this additional constraint the field `fixed` is added to the class `Fig`. For fixed figures the value of this field will be 1 and 0 otherwise. The code for the class `Point` is the same as in Figure 1. To implement the constraint the handler `Enforce` skips invoking the base code whenever the figure is fixed (checked by accessing the field `fixed`).

```
1  class Fig extends Object{ int fixed; }
2  Fig event Changed {
3   Fig fe;
4   provides fe != null
5   requires {
6    if(fe.fixed == 0){
7     invoke(next)
8     preserve fe==old(fe)
9    } else {
10    preserve fe==old(fe)
11   }
12  }
13 }
14 class Enforce extends Object {
15  when Changed do check;
16  /* ... */
17  Fig check(thunk Fig rest, Fig fe){
18   if(fe.fixed == 0){
19    invoke(rest)
20   } else {
21    refining preserve fe==old(fe){
22      fe
23    }
24   }
25  }
26 }
```

Figure 6: Specifying narrowing with a translucid contract

The contract for the event type `Changed` documents the possibility of a narrowing handler on lines 5–11. It does not, however, reveal the actual code of the narrowing handler as long as the hidden code refines the specification on line 8 and 10.

All observer's handler of the event type `Changed` must refine its specification. This means that the implementation of such handlers must structurally match the contract on lines 6–11. The implementation of the handler `Enforce` structurally matches the contract thus it structurally refines it. The true block of the **if** expression on line 18–20 refines the true block of the if expression in the specification on lines 6–9 because the empty expression trivially preserves `fe==`**old**`(fe)`. The false block of the **if** expression on line 20–24 refines the false block of the **if** expression in the specification on lines 9–11 because lines 21–23 claim to refine the specification.

3.3 Direct Interference: Replacement

A replacement handler omits the execution of the original event body and instead only runs the handler body. In Ptolemy this can be achieved by omitting the **invoke** expression in the handler, equivalent to not calling **proceed** in an around advice in AspectJ.

Figure 7 shows an example of such handler. The example uses several standard sugars such as += and > for ease of presentation. In this example, the method `moveX` causes a point to move along the x-axis by amount `d`. The handler `scaleit` implements the requirement that the "amount of movement should be scaled by a

```
1  class Point extends Fig {
2   int x; int y;
3   Fig moveX(int d){
4    announce Moved(this,d){
5     this.x += d; this
6    }
7   }
8  }
9  Fig event Moved {
10  Point p;
11  int d;
12  provides p!=null &&
13           d>0
14  requires {
15   preserve p!=null &&
16    p.y == old(p.y)
17  }
18 }
19 class Scale extends Object {
20  when Moved do scaleit;
21  int s; /*scale factor*/
22  Fig scaleit(thunk Fig rest,
23      Point p, int d){
24   refining preserve p!=null
25     && p.y == old(p.y) {
26     p.x += s * d ; p
27   }
28  }
29 }
```

Figure 7: Specifying replacement with a translucid contract

scaling factor `s` defined in class `Scale`". Specification of event type `Moved` documents that a replacement handler will be run when this event is announced by omitting the calls to **invoke** in its contract. The specification also documents the invariants maintained by the handlers that may run when the event is announced.

One requirement when writing translucid contracts is to reveal all calls to **invoke** expression. Therefore, if an event type's contract has no **invoke** expression, none of the event type's handlers are allowed to have an **invoke** expression in their implementation. Otherwise the structural similarity criterion of refinement is violated. The handler `scaleit` correctly refines `Moved`'s contract because its body (line 24-27) matches the specification. There are no invoke expressions and the invariant expected by the event type's contract (lines 15–16) and that maintained by the body (lines 24–25) are the same.

3.4 Direct Interference: Combination

Combination handlers can evaluate the **invoke** expression any number of times. In AspectJ, this would be equivalent to one or more calls to **proceed** in an around advice, guarded by some condition or in a loop. A combination handler is typically useful for implementing functionalities like fault tolerance. We show an example of a combination handler in Figure 8. In this example, we extend figures in the drawing editor to have colors. This is done by adding a field color to class `Fig` and by providing a method `setColor` for picking the color of the figure. The class `Color` is not shown in the listing. It provides a method `nextCol` to get the next available color.

To illustrate combination, let us consider the requirement that "each figure should have a unique color". To implement this requirement, an event type `ClChange` is declared as an abstraction of events representing colors changes. The method `setColor` changes colors so it announces the event `ClChange` on lines 5–7. The body of this announce expression contains the code to obtain the next color (line 6). The handler `Unique` implements the uniqueness requirement by storing already used colors in a hash

```
1  class Fig {
2   Color c;
3   int colorFixed;
4   Color setColor(){
5    announce ClChange(this){
6     this.c = c.nextCol()
7    }
8   }
9  }
10 Color event ClChange{
11  Fig fe;
12  provides fe!=null
13  requires {
14   while(fe.colorFixed==0){
15    invoke(next);
16    either{
17     preserve fe != null
18    } or {
19     preserve fe != null
20    }
21   }
22  }
23 }
```

```
24 class Unique {
25  HashMap colors;
26  when ClChange do check;
27  Color check(thunk Color rest,
28                    Fig fe){
29   while(fe.colorFixed==0){
30    invoke(rest);
31    if(colors.get(fe.c)!=null){
32     refining preserve fe!=null{
33      colors.put(fe.c);
34      fe.colorFixed = 1;fe.c
35     }
36    }else{
37     refining preserve fe!=null{
38      fe.c
39     }
40    }
41   }
42  }
43 }
```

Figure 8: Specifying combination with a translucid contract

table (`colors`). The field `colorFixed` is also added to figure class to show that a unique color has been chosen and fixed for the figure. The initial value of this field is zero. When the handler method `check` is run it checks `colorFixed` to see if a color has been chosen for figure or not, if not it then invokes the event body generating the next candidate color for the figure. If this color is already used, checked by looking it up in the hash table, event body is invoked again to generate the next candidate color. Otherwise, the current color is inserted in the hash table and `colorFixed` is set to one.

The specification for the event type `ClChange` documents that a combination handler will be run when this event is announced. This specification makes use of our novel feature, the choice feature, on line 16–20. To correctly refine this specification, a handler can either have a corresponding **`if`** expression at the corresponding place in its body or it may have an expression that runs unconditionally and refines the **`either`** block or the **`or`** block in the specification. By analyzing the specification, specially by while loop revealed in the specification, the base code developers can understand that the handlers that run when the event `ClChange` is announced may run the original event body multiple times. They are, however, not aware of the concrete details of such handlers, thus those details remain hidden. Since the handler `Unique`'s body structurally matches the specification, it correctly refines the specification.

3.5 More Expressive Control Flow Properties

Rinard *et al.*'s control flow properties are only concerned about calls to **`invoke`**. Their proposed analysis technique can decide which class of interference and category of control effects each isolated advice belongs to [44]. However, it could not be used to analyze possibility of two or more control flow paths each of which is an augmentation, however, each path maintains a different invariant. Figure 9 illustrates such an example. This example is adapted from the work of Khatchadourian and Soundarajan [26].

The class `Fig` not shown here is the same as in Figure 1. Khatchadourian and Soundarajan [26] implement an additional requirement that "a point should always be visibly distinguished from the origin". To implement this requirement a scaling factor `s` is added as a field to the class `Point` (line 2). This factor is initially set to 1 (line 5). The additional requirement is implemented as the class `Scaling`. The handler method `scale` in this class is run when event `Moved` is announced. The handler method ensures that if the point is close enough to the origin (vicinity condition) to visibly distinguish it from the origin the scaling factor is set to 10. Thus the scaling factor only gets two values 1 or 10. The vicinity condition is true if point's x and y coordinates are less than 5.

```
1  class Point extends Fig{
2   int x; int y; int s;
3   Point init(int x,int y){
4    this.x=x; this.y=y;
5    this.s=1; this
6   }
7   int getX(){this.x*this.s}
8   int getY(){this.y*this.s}
9   Fig move(int x, int y){
10   announce Moved(this){
11    this.x=x;this.y=y; this
12   }
13  }
14 }
15 Fig event Moved{
16  Point p;
17  provides p!=null
18  requires{
19   invoke(next);
20   if(p.x<5&& p.y<5){
21    establish p.s==10
22   } else {
23    establish p.s==1
24   }
25  }
26 }
27 class Scaling extends Object{
28  when Moved do scale;
29  Fig scale(thunk Fig rest, Point p){
30   invoke(rest);
31   if(p.x<5 && p.y<5){
32    refining establish p.s==10{
33     p.s=10; p
34    }
35   } else {
36    refining establish p.s==1{
37     p.s == 1; p
38    }
39   }
40  }
41 }
```

Figure 9: Expressive Control Flow Properties Beyond [44]

The assertions we want to validate in this example are as follows: (i) all of the handlers are after augmentation handlers, (ii) the value of scaling factor *s* is either 1 or 10, and (iii) the scaling factor s it is set to 10 if and only if the vicinity condition holds. Rinard *et al.*'s proposal could be used to verify (i) and a behavioral contract can specify (ii) but none of them could specify (iii), whereas our approach can. On lines 19–24 is a specification that conveys to the developers of the class `Point` that a conforming handler method will satisfy all three assertions above.

In summary, in this section we have shown that translucid contracts allow us to specify control flow interference between subject and observers. Specified interference patterns are enforced automatically through refinement. We are able to specify and enforce control interference properties proposed by Rinard *et al.*. There are more sophisticated control flow interplay patterns which could not be enforced by the previous work on design by contract for aspects whereas it could be specified as translucid contracts.

4. APPLICABILITY TO OTHER APPROACHES FOR AO INTERFACES

In this section, we discuss the applicability of our technique to other approaches for AO interfaces. As discussed previously, there are several competing and often complementary proposals

for AO interfaces. For example, Kiczales and Mezini's aspect-aware interfaces (AAI) [29], Gudmundson and Kizales pointcut interfaces [20], Sullivan *et al.*'s crosscutting interfaces (XPIs) [49], Aldrich's Open Modules [1], Steimann *et al.*'s joinpoint types [47], and Rajan and Leavens's event types [41]. We have tried out several of these ideas and our approach works beautifully. Since Steimann *et al.*'s joinpoint types [47] and Hoffman and Eugster's explicit joinpoints (EJP) are similar in spirit to Rajan and Leavens's event types [41], which we have already discussed in previous sections, we do not present our adaptation to these ideas here. Rather we focus on the AspectJ implementation of the XPI approach [52] and Aldrich's Open Modules [1] that are substantially distinct from event types [41, Fig. 10].

4.1 Translucid Contracts for XPIs and AAIs

Sullivan *et al.* [49] proposed a methodology, that they call crosscut programming interface (XPI) for aspect-oriented design based on design rules. The key idea is to establish a design rule interface which serves to decouple the base design and the aspect design. These design rules govern exposure of execution phenomena as joinpoints, how they are exposed through the joinpoint model of the given language, and constraints on behavior across joinpoints (e.g. provides and requires conditions [52]). XPIs prescribe rules for joinpoint exposure, but do not provide a compliance mechanism. Griswold *et al.* have shown that at least some design rules can be enforced automatically using AspectJ's features [52]. Current proposals for XPIs, however, all use behavioral contracts [49]. As shown previously, use of behavioral contracts, limits the expressiveness of the assertions which could be made using XPI. Behavioral contracts could not reveal implementation details which might be needed for some assertions [12].

```
1  class Fig extends Object{ int fixed; }
2  aspect XChanged {
3   pointcut joinpoint(Fig fe): target(fe)
4              && call(void Fig+.set*(..));
5   provides fe != null
6   requires {
7    if(fe.fixed == 0){
8     proceed();
9     preserve fe==old(fe)
10   } else {
11    preserve fe==old(fe)
12   }
13  }
14 }
15 aspect Enforce {
16  Fig around (Fig fe):
17      XChanged.joinpoint(fe){
18   if(fe.fixed == 0){
19    proceed()
20   } else {
21    refining preserve fe==old(fe){
22      fe
23    }
24   }
25  }
26 }
```

Figure 10: Applying translucid contract to an XPI

In this section, we show that translucid contracts can also be applied to enable expressive assertions about aspect-oriented programs that use the XPI approach. We also discuss changes in the refinement rules that is needed to verify such programs. To illustrate consider the example from Section 3.2, where constraint on movement of figures is implemented as an XPI and an aspect. An XPI typically also contains a description of scope, which we omit here. In the context of XPIs, the language for expressing translucid contract is slightly adapted to use **`proceed`** instead of **`invoke`** on line 8. Other than this change, our syntax for translucid contracts works right out-of-the-box.

Unlike translucid contracts for event types in Ptolemy, where the contract is thought to be attached to the type, in the AspectJ's version of XPI contracts are thought to be attached to the pointcut declaration, e.g. the contract on lines 5–13 is attached to the pointcut on lines 3–4. The variables that can be named in the contract are those exposed by the pointcut. For example, the contract can only use `fe`.

Our proposal for verifying refinement also needs only minor changes. Unlike Ptolemy, where the event types of interest are specified in the binding declarations, in AspectJ's version of XPI, aspects reuse the pointcut declarations from the XPI in the advice declaration (lines 16–17). Our refinement rules could be added here in the AO type system. So for an advice declaration to be well-formed, its pointcut declaration must be well-formed, the advice body must be well-formed, and the advice body must refine the translucid contract of the pointcut declaration. This strategy works for basic pointcuts, for compound pointcuts constructed using rules such as (`pcd and pcd'` or `pcd` **`or`** `pcd'`), where both `pcd` and `pcd'` are reused from different XPIs and thus may have independent contracts more complex refinement rules will be needed, which we have not explored in this paper.

Joinpoint interfaces like XPI could be computed from the implementation rather than being explicitly specified given whole-program information. Kiczales and Mezini [29] follow this approach to extract aspect-aware interfaces (AAI). A detailed discussion of the trade-offs of such interfaces is the subject of previous work [42, 49]. However, an important property of AAIs is that advised joinpoints contain the details of the advice. An example based on the example in Figure 10 is shown in Figure 11. The extracted AAI for the method `setX` is shown on lines 3-4. An adaptation of this extraction to include translucid contracts will be to carry over the contract from the pointcut to the joinpoint shadow as shown on lines 5–13.

```
1  Point extends Fig {
2   int x; int y;
3   Fig setX(int x): Update -
4    after returning Update.joinpoint(Fig fe)
5     provides fe != null
6     requires {
7      if(fe.fixed == 0){
8        proceed();
9        preserve fe==old(fe)
10     } else {
11       preserve fe==old(fe)
12     }
13    }
14  /* body of setX */
15  }
```

Figure 11: Applying translucid contract to an AAI

The syntax and refinement rules similar to XPI are applicable here. Like AAI annotations that provide developers of `Point` with information about potentially advising aspects, added contract would provide developers of `Point` with richer abstraction over the aspect's behavior. Similar ideas can also be applied to aspect-oriented development environments such as AJDT, which provide AAI-like information at joinpoint shadows in an AO program.

4.2 Translucid Contracts for Open Modules

Aldrich's proposal on Open Modules [1] is closely related to Ptolemy's quantified, typed events [41]. Open modules allows a class developer to explicitly expose pointcuts for behavioral modifications by aspects, which is similar to signaling events using the **announce** expressions in Ptolemy. The implementations of these pointcuts remain hidden from the aspects. As a result, the impact of base code changes on the aspect is reduced. However, quantification in Ptolemy is more expressive compared to Open Modules. In open modules, each explicitly declared pointcut has to be enumerated by the aspect for advising. On the other hand, Ptolemy's quantified, typed events significantly simplify quantification. Instead of manually enumerating the joinpoints of interest, one can use the name of the event type for implicit non-syntactic selection of joinpoints. This affects applicability of translucid contracts to Open Modules.

```
1  module FigModule {
2   class Fig;
3   friend Enforce;
4   expose: target(fe) &&
5    call(void Fig+.set*(..));
6    provides fe != null
7    requires{
8     if(fe.fixed == 0){
9      proceed();
10     preserves fe == old(fe)
11    } else {
12     preserves fe == old(fe)
13    }
14   }
15 }
16 aspect Enforce {
17  Fig around (Fig fe): target(fe) &&
18   call(void Fig+.set*(...)){
19    if(fe.fixed == 0){
20     proceed()
21    } else {
22     refining preserve fe==old(fe){
23     fe
24    }
25   }
26  }
27 }
```

Figure 12: Applying translucid contract to Open Modules [39]

To show the applicability of translucid contracts to Open Modules, we revisit the narrowing example from Section 3.2. Figure 12 shows the implementation of the same scenario using Open Modules. In implementing the example, we use the syntax from the work of Ongkingco *et al.* [39] to retain similarity with other examples in this work. In the listing constraints on the movement of figure is encapsulated in the module `Enforce`. Open module `FigModule` exposes a pointcut of **`class`** `Fig` on lines 4–5, marked by the keyword **`expose`**. Exposed pointcut is advisable only by the **`friend`** aspect `Enforce`. Transcluid contract on lines 6–14 states the behavior of interaction between specified friend aspect and the exposed pointcut. The adaptations in the syntax of contracts are the same as in the case of XPI discussed in Section 4.1.

Like contracts in XPI, contracts in Open Modules are attached to a pointcut declaration, e.g. the contract on lines 6–14 is attached to the exposed pointcut defined on lines 4–5. The variables that can be named in the contract are those exposed by the pointcut. For example, the contract on lines 6–14 can only use the variable `fe`.

Proposed verifying refinement rules need to be modified slightly as well. In Ptolemy, event type of interest is specified in the binding declaration whereas in AspectJ's version of Open Modules, aspects could not reuse pointcuts exposed by an Open Module and need to enumerate the pointcut in the advice declaration again (lines 17–18). Our refinement rules could be added here in AO type system. Well-formedness of basic and compound pointcuts follow the same rules laid out in Section 4.1.

This example illustrates how our approach might be used as a specification and verification technique for Open Modules. The only challenge that we saw in this process was to match an aspect's pointcut definition with the open module's pointcut definition to import its contract for checking refinement. Like translucid contracts for Ptolemy, in the case of Open Modules specification serves as a more expressive documentation of the interface between aspects and classes.

5. RELATED IDEAS

There is a rich and extensive body of ideas that are related to ours. Here, we discuss those that are closely related under three categories: contracts for aspects, proposals for modular reasoning, and verification approaches based on grey box specifications.

Contracts for Aspects. This work is closest in spirit to the work on crosscut programming interfaces (XPIs) [23, 52]. XPIs also allow contracts to be written as part of the interfaces as **`provides`** and **`requires`** clauses. Similar to translucid contracts, the **`provides`** clause establishes a contract on the code that announces events, whereas the **`requires`** clauses specifies obligations of the code that handles events. However, the contracts specified by these works are mostly informal and cannot be automatically checked. Furthermore, these works do not describe a verification technique and contracts could be bypassed.

Skotiniotis and Lorenz [33, 46] propose contracts for both objects and aspects in their tool *Cona*. Rinard *et al.* [44] classify the interaction of advice and method into direct and indirect interactions. Direct interactions focus on control flow elements while indirect interactions are concerned about data elements. Each of direct and indirect interactions are further categorized into different classes of interactions. They have developed an analysis system that categorizes aspects and method interactions. Their classification and analysis system serves reasoning purposes. As an analysis system, it expect developers to enforce desired properties by informing them about classes that each aspect-method interaction belongs to. There is no specification/enforcement mechanism supported in their approach. Zhao and Rinard [55] propose Pipa as a behavioral specification language for AspectJ. Pipa supports specification inheritance and specification crosscutting. It relies on textual copying of specifications for specification inheritance and syntactical weaving of specification for specification crosscutting. Annotated AspectJ program with JML-like Pipa's specifications is transformed to JML and Java code. JML-based verification tool could be used later to enforce specified behavioral constraints. All of these ideas use behavioral contracts and thus may not be used to reason about control effects of advice.

Modular Reasoning. There is a large body of work on modular reasoning about AO programs on new language designs [1, 14, 17, 21, 25], design methods [23, 30, 32, 52], and verification techniques [3, 24, 31, 40]. Our work complements ideas in the first and the second category and can use ideas in the third category for improved expressiveness. Compared to work on reasoning about implicit invocation [4, 8, 19, 54], our approach based on structural refinement is significantly lightweight. Furthermore, it accounts for quantification that these ideas do not.

Oliveira *et al.* [38] introduce a non-oblivious core language with explicit advice points and explicit advice composition requiring effects modeled as monads to be part of the component interfaces. Their statically typed model could enforce control and data flow in-

terference properties. Their work shares commonalities with ours in terms of explicit interfaces having more expressive contracts to state and enforce the behavior of interactions. However, it is difficult to adapt their ideas built upon their non-AO core language, to II, AO, and Ptolemy as they do not support quantification.

Hoffman and Eugster Explicit Joint Points [21] and Steimann *et al.*'s Joint Point Types [47] share similar spirit with Rajan and Leavan's event types [41]. Although Steimann *et al.* proposed informal behavioral specification, but there is no explicit notion of formally expressed and enforced contracts, stating interactions behavior, in any of these approaches.

Grey Box Specification and Verification. This work builds upon previous research on grey box specification and verification [12]. Among others, Barnett and Schulte [6, 7] have considered using grey box specifications written in AsmL [5] for verifying contracts for .NET framework, Wasserman and Blum [53] also use a restricted form of grey box specifications for verification, Tyler and Soundarajan [51] and most recently Shaner, Leavens, and Naumann [45] have used grey box specifications for verification of methods that make mandatory calls to other dynamically-dispatched methods. Rajan *et al.* have used grey box specifications to enable expressive assertions about web-services [43]. Compared to these ideas, our work is the first to consider grey box specifications as a mechanism to enable modular reasoning about code that announces events from the code that handles events, which is a common idiom in AO and II languages.

6. FUTURE WORK

Having laid out the basic ideas behind translucid contracts we now turn to the remaining work necessary to incorporate translucid contracts in Ptolemy and other AO languages. First task would be to define a precise formalization of when translucid contracts are refined by the handler body and second would be give definition of how reasoning about **`announce`** expressions proceeds. We discuss our ideas to solve these problems in some detail below.

6.1 Checking Contract Refinement

In Ptolemy a module announcing an event of type p may assume that all handler methods that run when p is announced refine the translucid contract of p. Each such handler method then in turn must guarantee that it correctly refines the contract. Informally, showing refinement between a contract and a handler method builds on the notion of structural refinement from the work of Shaner *et al.* [45]. New ideas will be in the refinement of **`invoke`** and **`either`** `{..}` **`or`** `{..}` expressions. The contract may include specification notations (se) whereas implementations can only contain executable code, built solely from the program constructs (e) described in Figure 3. A handler method's implementation refines a translucid contract if it meets two criteria: first, that the code shares its structure with that given in the specification and second, that the body of every **`refining`** expression obeys the specification it is refining.

6.2 Reasoning about Announce Expressions

A technique for verification of the code that announces events is an important future work. The basic plan is to use the copy rule [35] substituting the abstract program that is the specification of the event type for the announce expression. This will be sound, due to structural refinement.

The technical difficulty is that handlers are essentially mutually recursive, since they can use **`invoke`** expressions to proceed to the next handler that applies to the event. Thus in the verification, it will be difficult to know, statically, how many times to substitute (unfold) the abstract program for **`invoke`** when reasoning about an announce expression. One possibility is to use predicates that describe the state of the active object list, so that in the verification one can do a case split, based on what handlers actually do apply to a particular announce expression at that point in the program.

7. CONCLUSION

Many recent proposals for aspect-oriented interfaces [1, 20, 21, 29, 41, 47, 49] show promise towards improving modularity of AO programs. An important benefit of these proposals is that they provide a syntactic location to specify contracts between the advised and the advising code. However, behavioral contracts [49, 55] are largely insufficient to reason about potentially important control-flow related properties of the advising code.

We show that translucid contracts that are based on grey box specifications [10, 11] are useful for understanding and enforcing such properties for Ptolemy programs. A handler method conforms with such contract, if its body refines the contract. All handler methods for an event type p are required to conform with the contract for p. We show how verification proceeds for code announcing events. Checking the handlers and the announce expressions only require module-level analysis. We also demonstrate the applicability of translucid contracts to other type of AO interfaces.

Besides direction already discussed in Section 6, adding translucid contracts to other AO compilers, integrating it with rich specification features in JML, and trying out larger examples would also be part of the future activities.

8. REFERENCES

[1] J. Aldrich. Open modules: Modular reasoning about advice. In *ECOOP'05*.

[2] R.-J. Back and J. von Wright. *Refinement Calculus: A Systematic Introduction*. Graduate Texts in Computer Science. Springer-Verlag, 1998.

[3] P. F. Baldi, C. V. Lopes, E. J. Linstead, and S. K. Bajracharya. A theory of aspects as latent topics. In *OOPSLA*, pages 543–562, 2008.

[4] L. Baresi, C. Ghezzi, and L. Mottola. On accurate automatic verification of publish-subscribe architectures. In *ICSE'07*.

[5] M. Barnett and W. Schulte. The ABCs of specification: AsmL, Behavior, and Components. *Informatica*, 25(4):517–526, November 2001.

[6] M. Barnett and W. Schulte. Spying on components: A runtime verification technique. In *SAVCBS*, 2001.

[7] M. Barnett and W. Schulte. Runtime verification of .NET contracts. *JSS*, 65(3):199–208, March 2003.

[8] J. S. Bradbury and J. Dingel. Evaluating and improving the automatic analysis of implicit invocation systems. In *ESEC/FSE'03*.

[9] M. Büchi. Safe language mechanisms for modularization and concurrency. Technical Report TUCS Dissertations No. 28, Turku Center for Computer Science, May 2000.

[10] M. Büchi and E. Sekerinski. Formal methods for component software: The refinement calculus perspective. In *WCOP'97*.

[11] M. Büchi and W. Weck. A plea for grey-box components. Technical Report 122, Turku Center for Computer Sc., 1997.

[12] M. Büchi and W. Weck. The greybox approach: When blackbox specifications hide too much. Technical Report 297, Turku Center for Computer Science, August 1999.

[13] C. Clifton. A design discipline and language features for modular reasoning in aspect-oriented programs. Technical Report 05-15, Iowa State University, Jul 2005.

[14] C. Clifton and G. T. Leavens. Obliviousness, modular reasoning, and the behavioral subtyping analogy. In *SPLAT'03*.

[15] C. Clifton and G. T. Leavens. MiniMAO$_1$: Investigating the semantics of proceed. *SCP*, 63(3):321–374, 2006.

[16] C. Clifton, G. T. Leavens, and J. Noble. Ownership and effects for more effective reasoning about Aspects. In *ECOOP '07*.

[17] D. S. Dantas and D. Walker. Harmless advice. In *POPL'06*.

[18] M. Flatt, S. Krishnamurthi, and M. Felleisen. A programmer's reduction semantics for classes and mixins. In *Formal Syntax and Semantics of Java*, pages 241–269, 1999.

[19] D. Garlan, S. Jha, D. Notkin, and J. Dingel. Reasoning about implicit invocation. In *FSE'98*.

[20] S. Gudmundson and G. Kiczales. Addressing practical software development issues in AspectJ with a pointcut interface. In *Advanced Separation of Concerns*, 2001.

[21] K. J. Hoffman and P. Eugster. Bridging Java and AspectJ through explicit join points. In *PPPJ'07*.

[22] A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In *OOPSLA '99*.

[23] K. J. Sullivan *et al.* Information hiding interfaces for aspect-oriented design. In *ESEC/FSE'05*.

[24] S. Katz. Diagnosis of harmful aspects using regression verification. In *FOAL'4*.

[25] A. Kellens, K. Mens, J. Brichau, and K. Gybels. Managing the evolution of aspect-oriented software with model-based pointcuts. In *ECOOP '06*.

[26] R. Khatchadourian and N. Soundarajan. Rely-guarantee approach to reasoning about aspect-oriented programs. In *SPLAT'07*.

[27] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *ECOOP'01*.

[28] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *ECOOP'07*.

[29] G. Kiczales and M. Mezini. Aspect-oriented programming and modular reasoning. In *ICSE'05*.

[30] G. Kiczales and M. Mezini. Separation of concerns with procedures, annotations, advice and pointcuts. In *ECOOP'05*.

[31] S. Krishnamurthi, K. Fisler, and M. Greenberg. Verifying aspect advice modularly. In *FSE'04*.

[32] C. V. Lopes and S. K. Bajracharya. An analysis of modularity in aspect oriented design. In *AOSD*, pages 15–26, 2005.

[33] D. H. Lorenz and T. Skotiniotis. Extending design by contract for aspect-oriented programming. *CoRR*, abs/cs/0501070, 2005.

[34] D. C. Luckham, J. J. Kennedy, L. M. Augustin, J. Vera, D. Bryan, and W. Mann. Specification and analysis of system architecture using Rapide. *TSE*, 21(4), Apr 1995.

[35] C. Morgan. Procedures, parameters, and abstraction: separate concerns. *Sci. Comput. Program.*, 11(1):17–27, 1988.

[36] C. Morgan. *Programming from Specifications: Second Edition*. Prentice Hall International, Hempstead, UK, 1994.

[37] J. M. Morris. A theoretical basis for stepwise refinement and the programming calculus. *SCP'87*.

[38] B. Oliveira, T. Schrijvers, and W. R. Cook. Effectiveadvice: Disciplined advice with explicit effects. In *AOSD'10*.

[39] N. Ongkingco, P. Avgustinov, J. Tibble, L. Hendren, O. de Moor, and G. Sittampalam. Adding Open Modules to AspectJ. In *AOSD'6*.

[40] K. Ostermann. Reasoning about aspects with common sense. In *AOSD*, 2008.

[41] H. Rajan and G. T. Leavens. Ptolemy: A language with quantified, typed events. In *ECOOP'08*.

[42] H. Rajan and K. J. Sullivan. Unifying aspect- and object-oriented design. *TOSEM*, 2008.

[43] H. Rajan, J. Tao, S. M. Shaner, and G. T. Leavens. Tisa: A language design and modular verification technique for temporal policies in web services. In *ESOP'09*.

[44] M. Rinard, R. Salcianu, and S. Bugrara. A classification system and analysis for aspect-oriented programs. In *FSE'04*.

[45] S. M. Shaner, G. T. Leavens, and D. A. Naumann. Modular verification of higher-order methods with mandatory calls specified by model programs. In *OOPSLA'07*.

[46] T. Skotiniotis and D. H. Lorenz. Cona: Aspects for contracts and contracts for aspects. In *OOPSLA'04*.

[47] F. Steimann, T. Pawlitzki, S. Apel, and C. Kastner. Types and modularity for implicit invocation with implicit announcement. *TOSEM*, 20(1), 2010.

[48] M. Stoerzer and J. Graf. Using pointcut delta analysis to support evolution of aspect-oriented software. In *ICSM'05*.

[49] K. J. Sullivan, W. G. Griswold, H. Rajan, Y. Song, Y. Cai, M. Shonle, and N. Tewari. Modular aspect-oriented design with XPIs. *TOSEM*, 20(2), 2009.

[50] T. Tourwé, J. Brichau, and K. Gybels. On the existence of the AOSD-evolution paradox. In *SPLAT'03*.

[51] B. Tyler and N. Soundarajan. Black-box testing of grey-box behavior. In *FATES'03*.

[52] W. G. Griswold *et al.* Modular software design with crosscutting interfaces. *IEEE Software'06*.

[53] H. Wasserman and M. Blum. Software reliability via run-time result-checking. *J. ACM*, 44(6):826–849, 1997.

[54] H. Zhang, J. S. Bradbury, J. R. Cordy, and J. Dingel. Using source transformation to test and model check implicit-invocation systems. *SCP'06*.

[55] J. Zhao and M. Rinard. Pipa: A behavioral interface specification language for AspectJ. In *FASE'03*.

A Smooth Combination of Role-based Language and Context Activation

Tetsuo Kamina
The University of Tokyo
7-3-1 Hongo, Bunkyo-ku, Tokyo
113-0033 Japan
kamina@acm.org

Tetsuo Tamai
The University of Tokyo
3-8-1 Komaba, Meguro-ku, Tokyo
153-8902 Japan
tamai@acm.org

ABSTRACT

In this paper, we propose a programming language called NextEJ. NextEJ is a smooth combination of a role-based language EpsilonJ and context activation mechanisms provided by COP languages. It supports all the features of the Epsilon model such as dynamic object-role binding and unbinding, and encapsulation of collaboration of roles as a context that can be defined as a reusable module. Furthermore, NextEJ tackles typing problem of the Epsilon model by introducing the *context activation scope* inspired by COP languages. The key ideas described in this paper are formalized as a small core language FEJ that is built on top of FJ. FEJ's type system is proven to be sound.

Categories and Subject Descriptors

D.1.5 [**Programming Techniques**]: Object-Oriented Programming; D.3.1 [**Programming Languages**]: Formal Definitions and Theory; D.3.3 [**Programming Languages**]: Language Constructs and Features

General Terms

Languages

Keywords

Role model, Epsilon, Context-oriented Programming

1. INTRODUCTION

Context-awareness is becoming an increasingly important feature in many types of applications, ranging from business applications to mobile and ubiquitous computing systems. For example, in location-based systems, the behaviors of the provided services are situation-dependent or even deeply personalized [6]; thus, instance-specific behavioral changes with respect to the surrounding context are required. Unfortunately, current mainstream object-oriented languages provide little explicit support for context-awareness [23].

To explicitly support context-awareness at the programming language level, several approaches have been proposed. One of the promising methods of realizing context-awareness is to use the *Epsilon model* [41, 42]. The Epsilon model provides a clear conceptual framework of role modeling and object adaptation to collaboration fields between roles. It also provides a good basis for developing context-aware applications. A Java-based implementation language called *EpsilonJ* has also been implemented [34].

In EpsilonJ, each context is declared by a context declaration statement (Figure 1). Each context consists of a set of roles that represents collaborations performed in that context. For example, Figure 1 shows collaborations between employers and employees in a company (e.g., an employer pays salary for the employees). Each context can be instantiated by using the `new` expression, as in Java. An object can participate in a context by assuming one of the role instances belonging to that context. This can be achieved by the `newBind` predefined method (Figure 2); this method creates a role instance and binds it with the object that is passed as an argument to `newBind`. An object can also assume other role instances belonging to other contexts.

EpsilonJ supports the development of context-aware applications. We can define behavioral variations that may vary with respect to the surrounding environment by using context declarations. In each context, we can group related context-dependent behaviors. Each context and role is a first-class entity that can be explicitly referred to by its name; thus, we can explicitly invoke context-dependent behaviors. However, in EpsilonJ, dynamically acquired methods obtained by assuming roles have to be invoked by using down-casting (line 6 of Figure 2); this is an unsafe operation. This down-casting is cumbersome and error-prone, because we always have to use it whenever we want to use role methods on a case-by-case basis. In addition, to assure that the receiver object is actually bound with the role, we have to

```
context Company {
  role Employer {
    void pay() { Employee.getPaid(); } }
  role Employee {
    int save, salary;
    Employee(int salary) {
      this.salary = salary; }
    void getPaid() { save += salary; } } }
```

Figure 1: Example of context in EpsilonJ

```
1 Person tanaka = new Person();
2 Person komiyama = new Person();
3 Company todai = new Company();
4 todai.Employer.newBind(komiyama);
5 todai.Employee.newBind(tanaka,1000);
6 ((todai.Employer)komiyama).pay();
```

Figure 2: Object-role binding in EpsilonJ

investigate the source code carefully.

One of the reasons for this problem is the fact that there is no method to control the scope of object-role binding. If we can represent a scope that ensures that the designated objects and roles are bound, this problem would be solved. This scoping mechanism is similar to the one that has been described in *Context-oriented Programming (COP)* [23, 13, 22]. However, COP languages do not provide the object adaptation mechanism supported by EpsilonJ because the binding mechanism is not object-based but class-based. Furthermore, in most the statically typed COP languages, such binding is performed statically and only the *activation* mechanism is provided. Therefore, the flexible adaptive evolution provided by EpsilonJ is not supported by COP languages.

We have proposed a smooth combination of the Epsilon model and COP languages that incorporates the advantages of both. We design a new programming language called NextEJ [28][1] that incorporates the feature of both Epsilon model and COP. NextEJ addresses the problem of the type unsafety of the Epsilon model by introducing a language feature, the concept of which is adopted from COP languages, called *context activation scope*. In the context activation scope, we can denote which role of an object that belongs to an context is bound and activated within the scope. If the designated object is not bound with the role, the role instance is implicitly created and bound with the object, and therefore it is ensured that the object always assumes the role within the scope and no method-not-understood errors occur at run-time. Furthermore, context activation scopes can be nested so that multiple contexts can be activated at a time. A role instance has a pre-defined field `thisC` that refers to its enclosing context instance. In the case of multiple context activations, the reference of `thisC` is interpreted as a composite context whose behavior is determined by the order of activation.

To carefully investigate the type soundness of NextEJ, we develop FEJ, a core calculus that combines the features of EpsilonJ and COP. This formalization is built on top of Featherweight Java (FJ) [24] and its type system is proven to be sound. There have been only a few reports on the formalization of the Epsilon model and COP languages [27, 12], and there have been non on the combination of role-based languages and COP. The formalization described in this paper can also be considered as a theoretical basis for similar languages such as ObjectTeams [21]. We discuss the relationship between NextEJ and ObjectTeams in section 4.

2. NEXTEJ: SMOOTH COMBINATION OF EPSILON AND COP

This section describes how context-awareness can be easily expressed by NextEJ. A design sketch of NextEJ was presented in [28]. For the sake of simplicity, we have included this sketch in this paper.

[1]The syntax of NextEJ has been improved in this paper.

```
class Building {
  role Guest {
    void escape() { .. }
  }
  role Security {
    void notify() {
      Guest.escape(); }
  }
}
```

```
class Shop {
  role Customer {
    void buy(Item i) {
      int p = i.getPrice();
      Seller.getPaid(p);
    } }
  singleton role Shopkeeper {
    void getPaid(int price)
      { ... } } }
```

Figure 3: Context and role declarations

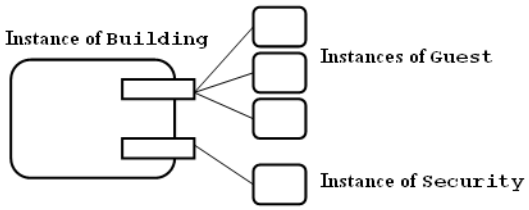


Figure 4: Structure of role instances and a context instance

2.1 An Example

To discuss the main features of NextEJ, we consider the following example. This example features two contexts, building and shop. Within a building, there exist several roles such as visitor, janitor, security agent, and owner. Similarly, within a shop there exist some roles such as customer and shopkeeper. When a person enters a building, she assumes the role of a visitor. Similarly, a person assumes the role of a customer when she enters a shop. There exist many interactions among roles; e.g., a security agent notifies all visitors in case of an emergency or a shopkeeper sells a customer an item. When a person leaves a context (e.g., building) she quits the role she assumes (e.g., visitor). Furthermore, shops may be within a building; therefore a person may simultaneously enter multiple contexts (i.e., building and shop).

2.2 Context and Role Declarations

NextEJ is an extension of Java that provides an explicit method to represent context-dependent behaviors and object adaptation to contexts. Figure 3 shows an example of context and role declarations in NextEJ. In NextEJ, each context is declared as a normal class and each role is declared within a class by using the `role` declaration statement. In Figure 3, the context `Building` consists of two roles, `Guest` and `Security`. Within roles, we can declare methods and fields. For example, the role `Guest` declares a method `escape()` that is called in the body of `notify()` declared in `Security`.

A context can be instantiated by a `new` expression because it is actually a normal class. On the other hand, an instance of a role cannot be created explicitly, as described later. The relationships among role instances and the enclosing context instance are shown in Figure 4. A role instance is always associated with an instance of its enclosing context. A set of

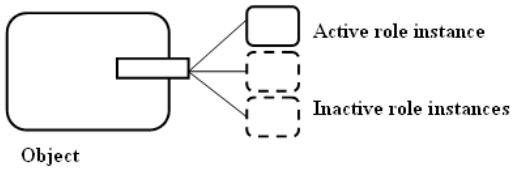


Figure 5: Structure of an object bound with role instances

instances of a role associated with the same enclosing context instance is called a role group, and it is referred by the role name. For example, the method call `Guest.escape()` is interpreted as calling the `escape()` methods of all `Guest` instances. A role declared as `singleton` is called a *singleton role*; this implies that at most one instance of the role with the same enclosing context instance can be created.

A role declaration cannot contain an `extends` clause; however, this does not mean that a role cannot extend other roles. The composition mechanism of contexts and roles is discussed in section 2.5.

2.3 Object Adaptation and Context Activation

A class instance enters a context by assuming one of its role instances. Furthermore, a class instance can be bound with multiple role instances and can activate or deactivate some of them (Figure 5). For example, assuming that we have a class `Person`, object adaptation to a context can be written as follows:

```
final Building midtown = new Building();
Person tanaka = new Person();
Person suzuki = new Person();
Person sato = new Person();
bind tanaka with midtown.Guest(),
     suzuki with midtown.Guest(),
     sato with midtown.Security() {
  ...
  sato.notify(); }
```

The sentence beginning from the keyword `bind` is called a *context activation scope*. Before entering the execution scope (enclosed between braces), it creates role instances and binds them with the corresponding class instances, if these class instances are not bound with the corresponding roles. If a class instance is already bound with the corresponding role, the role instance is not created but the existing role instance is *activated*. Within the parentheses following the role name, we specify the arguments for the constructor of the role. These arguments are used only when the class instance is not bound with the role so that the role instance is created.

After entering the execution scope, it is assumed that each class instance declared in the `bind` clause is bound with the corresponding role instance. For example, in the above code, `sato` is bound with a role `midtown.Security()` (implying that `sato` enters the context `midtown` as a `Security`). Within the following brace, `sato` acquires the behavior (and states) declared in `Building.Guest`; thus, we can safely call the method `notify()` declared in `Building.Guest` on `sato`. Within the context activation scope, it is considered that `sato` is a subtype of `Person` and `Building.Guest`, like multiple inheritance or mixins [9]. As described later, a context can also be composed with another context, and a subtyping relation exists between a context and the composite context. To ensure type safety, all the variables referring to a context instance have to be declared with the modifier `final`, because if a reference to a context instance changes during computation, it becomes very difficult to determine the actual type of role instances belonging to the context[2].

Note that outside the context activation scope, we cannot access methods declared in roles. This does not imply that the acquired role is discarded outside the scope. Instead, the role instance and its states are retained but *deactivated*, recovering the original behavior of the object. The retained role instance will be reactivated if the object assumes the same role in the same context.

The idea of activation/deactivation of role instances is taken from ContextJ and is one of the major differences from EpsilonJ. Within the context activation scope, it is always assumed that the object is bound with the corresponding role instance; thus, we can safely access the role instance method. In EpsilonJ, on the other hand, once an object is bound with a role instance, this role instance is activated only through down-cast expressions. Because whether an object is bound or unbound with the role instance cannot be determined statically, this down-casting may result in a cast-exception. Once the object is unbound with the role, the role instance becomes garbage. Instead, in NextEJ, the deactivated role instance may be activated again, preserving its states.

In NextEJ, the context activation is dynamically scoped, because role method invocation is based on role instances that are dynamically bound with a class instance, and this binding can go beyond the lexical scope (e.g., by passing the instance as an argument to a method invocation).

2.4 Multiple Context Activation

A class instance may enter multiple contexts. For example, there exists a case in which a shop is within a building; in this case, a customer of the shop is also a guest of the enclosing building. To represent such a situation (i.e., there are multiple contexts in which the person is participating), the context activation scope can be nested, as shown in the following example:

```
final Building midtown = new Building();
Person tanaka = new Person();
bind tanaka with midtown.Guest() {
  final Shop lawson = new Shop();
  Person sato = new Person();
  bind tanaka with lawson.Customer(),
       sato with lawson.Seller() {
    tanaka.buy(someItem); } ... }
```

In this example, `tanaka` first enters the context `midtown` as a `Guest`; then, it enters the context `lawson`, located within

[2]There may be other approaches to solve this problem (e.g., using *exact types*) [10, 25]; however, we prefer to use the approach of "final context instance," because it simplifies the type system and it is easy to reason about the correctness of the program. Furthermore, we observe that the reference to a context is inherently difficult to change (like aspects in AOP); thus, we are not sure when "non-final" context instances become useful.

```
class Building {
  String name;
  Building(String name) {
    this.name = name; }
  void currentPosition() {
    System.out.println(
      " "+name);
    next();
  }
  role Guest { ... }
  role Security { ... }
}
```

```
class Shop {
  String name;
  Shop(String name) {
    this.name = name; }
  void currentPosition() {
    System.out.println(
      " "+name);
    next();
  }
  role Customer { ... }
  role Seller { ... }
}
```

Figure 6: Context method combination

midtown, as a Customer; finally, it buys someItem (and pays to sato, as shown in Figure 3).

2.5 Referring the Enclosing Context and Composite Context

Another feature of NextEJ that is not provided by EpsilonJ is that the enclosing context instance and its methods can be accessed through the special field thisC that is implicitly declared in all role declarations and always refers to the enclosing context instance. Therefore, if contexts Building and Shop are declared as shown in Figure 6, the following code is allowed in NextEJ:

```
final Building midtown = new Building("Midtown");
Person tanaka = new Person();
bind tanaka with midtown.Guest() {
  final Shop lawson = new Shop("Lawson");
  bind tanaka with lawson.Customer() {
    tanaka.thisC.currentPosition(); } }
```

In this code, the field thisC is accessed on tanaka; this is allowed because tanaka is bound with role instances. Because the enclosing context instance declares a method currentPosition (that can be statically assured by using the information provided by the context activation scope), we can safely call currentPosition on thisC; this prints where tanaka reside on the standard output.

In addition, note that tanaka enters two contexts, midtown and lawson, both of which declare the method currentPosition. Actually, on thisC, we can access a composite context of midtown and lawson. The ordering of the composition is determined by the order of activation; the innermost context always precedes the other contexts. In Figure 6, the declaration of currentPosition contains a method call next() that is similar to inner of Beta [33, 16]. It calls the next method if it exists. If the next method does not exist, calling next() has no effects. Therefore, currentPosition declared in Shop is first called; then, that declared in Building is called. The above code therefore prints a string " Lawson Midtown" on the standard output.

If each component context of a composite context has a role with the same name, an access to the role name is also interpreted as a composite role, and the innermost role always precedes the other contexts. This mechanism is similar to family polymorphism [15]. Furthermore, a context can also extend another context. If both of a context and its derived context declare roles with the same name, the role in the derived context implicitly inherits from the role in the super context, and no subtyping is defined between them. This mechanism is similar to lightweight family polymorphism [40]. In the case of method name conflict (i.e., a role method overrides methods provided by both its super role and another component role), the super role method precedes the others (like nested inheritance [35, 36]).

2.6 Swapping Roles

As mentioned earlier, a role instance is deactivated outside the context activation scope. This deactivated role instance can be discarded[3]. Furthermore, as in EpsilonJ, another class instance may also assume the removed role instance. We can express it by using the bind statement (context activation scope) followed by the from clause:

```
Person sato = new Person();
bind sato with lawson.Seller() from tanaka {
  ... }
```

The above code results in tanaka dropping the instance of role lawson.Seller and sato taking it over (if tanaka is not bound with lawson.Seller, a new instance of it is created for sato).

2.7 Other Features Taken from EpsilonJ

NextEJ also has a few other features found in EpsilonJ. For example, a role may declare a *required interface*. This is a method of defining an interface to a role and it is used at the time of binding with a class instance, requiring the class instance to supply that interface, i.e., the binding class instance should possess all the methods specified in the interface. A required interface can be declared using the requires clause as follows:

```
class Building {
  role Guest requires { String name(); } {
    ... } }
```

When a required interface is declared to a role, methods can be imported from the binding class instance. For example, supposing that Person has a method name(), in the aforementioned bind statements, the method name() of tanaka is imported to the Guest role instance through the interface. The imported method can be used in the body of the role declaration. Furthermore, the role may override the imported method, and in the overriding method, we may call the original (overridden) method by calling the method with the same signature on super.

For type-checking this binding, it is only necessary for the class to have a method that has the same name and the same signature required by the role. In other words, the class has to be a *structural subtype* of the requires interface[4].

2.8 Summary

As discussed in [28], NextEJ supports all the basic requirements of COP languages. Furthermore, it provides a flexible mechanism for object adaptation that was originally proposed by the Epsilon model. In NextEJ, each context represents a concern so that separation of concerns is explicitly supported by the language. Unlike ContextJ's layer-in-class approach, contexts including roles can be units of reuse. Instance-specific context-dependent behaviors are supported

[3]This can be achieved using the unbind predefined method described in [28]

[4]A similar mechanism is also found in McJava, a Java extension with mixins [26].

$$
\begin{array}{rcl}
L & ::= & \texttt{class}\ C\ \{\ \bar{T}\ \bar{f};\ \bar{M}\ \bar{A}\} \\
A & ::= & \texttt{role}\ R\ \texttt{requires}\ \{\ \bar{M_I}\ \}\{\bar{T}\ \bar{f};\ \bar{M}\} \\
T & ::= & C.R \mid \bar{C}.\bar{R} :: C \\
T_S & ::= & T \mid \{\ \bar{M_I};\ \} \\
M_I & ::= & T\ m(\bar{T}\ \bar{x}) \\
M & ::= & M_I\ \{\ \texttt{return}\ e;\ \} \\
e & ::= & x \mid e.f \mid e.m(\bar{e}) \mid \texttt{new}\ C(\bar{e}) \oplus \bar{r} \mid \\
 & & \texttt{bind}\ \bar{x}\ \texttt{with}\ \bar{r}\ \texttt{from}\ \bar{y}\ \{\ \bar{x}\bar{y}.e_0\ \} \\
v & ::= & \texttt{new}\ C(\bar{v}) \oplus \bar{r} \\
r & ::= & v.R(\bar{v})
\end{array}
$$

Figure 7: Syntax

$$T_S <: T_S \qquad \text{(S-REFL)}$$

$$\frac{C <: D \qquad D <: E}{C <: E} \qquad \text{(S-TRANS)}$$

$$\frac{T\ m(\bar{T}\ \bar{x}); \in \bar{M_I} \Rightarrow mtype(m, C) = \bar{T} \to T}{C <: \{\bar{M_I}\}} \qquad \text{(S-STRUCT)}$$

$$C.R :: T <: T \qquad \text{(S-MIXINC)}$$

$$C.R :: T <: C.R \qquad \text{(S-MIXINR)}$$

Figure 8: Subtyping

so that within the context activation scope, objects that are not bound with the roles are not affected by the activation. As in EpsilonJ, the role-class binding is performed at run-time. While developing a context, NextEJ does not assume any existing code (i.e., we can design contexts independently), because the `requires` interface only imposes structural subtyping on roles and classes.

Furthermore, unlike EpsilonJ, NextEJ provides a mechanism for clearly defining the scope where the context-dependent behaviors are activated. This scoping mechanism ensures type-soundness. In the following section, we provide the formalization of key ideas presented in this paper.

3. FEJ: CORE CALCULUS OF NEXTEJ

In this section, we formalize the core features of NextEJ described in the previous sections as a small calculus called FEJ. NextEJ provides a number of interesting features. In this paper, we focus on the most relevant features for object adaptation and context-activation mechanisms. FEJ is built on top of Featherweight Java (FJ) [24], a functional core of class-based object-oriented languages such as Java.

3.1 Syntax

The abstract syntax of FEJ is shown in Figure 7. The metavariables C and D range over classes; S, T, U, and V range over named types; M ranges over method declarations; L ranges over class declarations; Q and R range over roles; A ranges over role declarations; M_I ranges over method signatures; f and g range over fields; m and n range over method names; T_S ranges over types (including interface types); b, c, d, and e range over expressions; x and y range over variables; r and s range over role instances; and v and w range over values.

We write $\bar{M}$ as a shorthand for a possibly empty sequence $M_1 \cdots M_n$, $\bar{A}$ as a shorthand for $A_1 \cdots A_n$, and $\bar{e}$ as a shorthand for $e_1, \cdots, e_n$. We also abbreviate pairs of sequences in a similar manner, writing $\bar{T}\ \bar{f}$; as a shorthand for $T_1\ f_1; \cdots T_n\ f_n;$, $\bar{T}\ \bar{x}$ as a shorthand for $T_1\ x_1, \cdots, T_n\ x_n$, and $\bar{C}.\bar{R}$ as a shorthand for $C_1.R_1 :: \cdots :: C_n.R_n$. We denote an empty sequence as $\cdot$ and the length of sequence $\bar{e}$ as $\#(\bar{e})$. Sequences are assumed to contain no duplicate names.

In FEJ, the body of class declaration consists of field declarations, followed by method declarations and role declarations. Similarly, the body of role declaration consists of field declarations followed by method declarations. The inheritance of classes is omitted in this calculus, implying that the family-polymorphism-like features of NextEJ is totally excluded on FEJ. Although this feature is interesting and very useful, it is not technically a new feature. To realize a clear understanding of our new features provided by NextEJ, FEJ is designed to be concentrated on the most relevant features of object adaptation and context activation.

A named type is a class name, a role name (prefixed by a class name), or a composite type of a sequence of roles and a class. Named types appear in field declarations, signature of method declarations and constructor declarations, and return type of method declarations. On the other hand, interface types $\{\ \bar{M_I}\}$ can only appear in the `requires` clause of role declarations. The special variable `super` within the role declaration is assumed to have an interface type appearing in the `requires` clause.

A method declaration consists of the method signature and a return type, followed by its body. The body of the method declaration consists of just one `return` statement, implying that FEJ is a purely functional calculus.

There are five kinds of expressions in FEJ: variables, field accesses, method invocations, class instance creations, and bind expressions. A class instance creation consists of normal Java's `new` syntax and its associated role instances, which may vary during computation. The body of a bind expression only consists of one expression. Within the body of the bind expression, variables appearing in the `bind` clause and `from` clause are not considered as free variables. To explicitly denote this fact, we write $\bar{x}\bar{y}.e_0$ to imply that $\bar{x}$ and $\bar{y}$ are bound in e_0. We assume that `this` and `super` are special variables that are implicitly declared in method declarations.

An FEJ program is a pair of a class table CT and an expression e. A class table is a map from class names to class declarations. The expression e may be considered as the `main` method of the real NextEJ program. The class table is assumed to satisfy the following conditions: (1) $CT(C) =$ `class` $C \cdots$ for every $C \in dom(CT)$ and (2) $C \in dom(CT)$ for every class name appearing in the range of CT. In the derivation hypothesis shown below, we abbreviate $CT(C) =$ `class` $C \cdots$ as `class` $C \cdots$.

Subtyping rules of FEJ are shown in Figure 8. In FEJ, subtyping is a reflective and transitive closure of the mixin composition (::) relation. Furthermore, a class is a subtype of an interface if the class implements all the methods declared in the interface; thus, there exists structural subtyping between classes and interfaces.

$$\frac{\texttt{class}\ C\ \{\ \bar{T}\ \bar{f};\ \bar{M}\ \bar{A}\}}{\mathit{fields}(C) = \bar{T}\ \bar{f}}$$

$$\frac{\texttt{class}\ C\ \{\ \bar{T}\ \bar{f};\ \bar{M}\ \bar{A}\} \quad \texttt{role}\ R\ \texttt{requires}\ \{\ \cdots\ \}\{\ \bar{S}\ \bar{g};\ \cdots\} \in \bar{A}}{\mathit{fields}(C.R) = \bar{S}\ \bar{g}}$$

$$\frac{\mathit{fields}(C) = \bar{T}\ \bar{f}}{\mathit{ftype}(f_i, C) = T_i}$$

$$\frac{\mathit{fields}(C.R) = \bar{T}\ \bar{f}}{\mathit{ftype}(f_i, C.R) = T_i}$$

$$\frac{\mathit{ftype}(f, C.R) = S}{\mathit{ftype}(f, C.R :: T) = S}$$

$$\frac{\mathit{fields}(C.R) = \bar{T}\ \bar{f} \qquad f \notin \bar{f}}{\mathit{ftype}(f, C.R :: T) = \mathit{ftype}(f, T)}$$

Figure 9: Field lookup

$$\frac{\texttt{class}\ C\ \{\ \bar{T}\ \bar{f};\ \bar{M}\ \bar{A}\ \}}{\mathit{fvalue}(f_i, \texttt{new}\ C(\bar{e}) \oplus \cdot) = e_i}\ \text{(FV-CLASS)}$$

$$\frac{\texttt{class}\ C\ \{\ \cdots \bar{A}\ \} \quad \texttt{role}\ R\ \texttt{requires}\ \{\ \cdots\}\{\ \bar{T}\ \bar{f}\ \cdots\ \} \in \bar{A}}{\mathit{fvalue}(f_i, \texttt{new}\ D(\cdots) \oplus \texttt{new}\ C(\bar{e}).R(\bar{d})\bar{r}) = d_i}\ \text{(FV-ROLE)}$$

$$\frac{\texttt{class}\ C\ \{\ \cdots \bar{A}\ \} \qquad f \notin \bar{f} \quad \texttt{role}\ R\ \texttt{requires}\ \{\ \cdots\}\{\ \bar{T}\ \bar{f}; \cdots\ \} \in \bar{A}}{\mathit{fvalue}(f, \texttt{new}\ D(\cdots) \oplus \texttt{new}\ C(\bar{e}).R(\bar{d})\bar{r}) = \mathit{fvalue}(f, \texttt{new}\ D(\cdots) \oplus \bar{r})}\ \text{(FV-ROLE1)}$$

Figure 10: Field value lookup

3.2 Auxiliary Definitions

For the typing and reduction rules, we require a few auxiliary definitions. Field lookup functions are defined in Figure 9. The function $\mathit{fields}(\_)$, where $\_$ is either C or $C.R$, is a sequence $\bar{T}\ \bar{f}$ of field types and names declared in C or $C.R$. The function $\mathit{ftype}(f, T)$ returns the type S of field f, if f can be accessed on T. We write $f \notin \bar{f}$ to imply that the field f is not included in $\bar{f}$. The definitions of both *fields* and *ftype* are straightforward.

Figure 10 shows field value lookup rules. The function $\mathit{fvalue}(f, v)$ returns the value w of field f, if f can be accessed on either $\bar{r}$ or v. It first searches for f on the sequence of role instances $\bar{r}$. In this searching, the fields declared in the role instance of the left-most side of $\bar{r}$ are searched, followed by the next role instance. If f is not found on $\bar{r}$, fields declared in the class instance $\texttt{new}\ C(\bar{e})$ are searched.

$$\frac{\texttt{class}\ C\ \{\ \bar{T}\ \bar{f};\ \bar{M}\ \bar{A}\} \quad T\ m(\bar{T}\ \bar{x})\{\ \texttt{return}\ e;\ \} \in \bar{M}}{\mathit{mtype}(m, C) = \bar{T} \to T}$$

$$\frac{\texttt{class}\ C\ \{\ \bar{T}\ \bar{f};\ \bar{N}\ \bar{A}\ \} \quad \texttt{role}\ R\ \texttt{requires}\ \{\bar{M}_I\}\{\cdots\ \bar{M}\} \in \bar{A} \quad T\ m(\bar{T}\ \bar{x})\{\ \texttt{return}\ e;\ \} \in \bar{M}}{\mathit{mtype}(m, C.R) = \bar{T} \to T}$$

$$\frac{\texttt{class}\ C\ \{\bar{T}\ \bar{f}; \bar{N}\ \bar{A}\} \quad \texttt{role}\ R\ \texttt{requires}\ \{\bar{M}_I\}\{\cdots\ \bar{M}\} \in \bar{A} \quad m \notin \bar{M} \qquad T\ m(\bar{T}\ \bar{x}) \in \bar{M}_I}{\mathit{mtype}(m, C.R) = \bar{T} \to T}$$

$$\frac{T\ m(\bar{T}\ \bar{x}) \in \bar{M}_I}{\mathit{mtype}(m, \{\ \bar{M}_I\ \}) = \bar{T} \to T}$$

$$\frac{\mathit{mtype}(m, C.R) = \bar{T} \to T}{\mathit{mtype}(m, C.R :: S) = \bar{T} \to T}$$

$$\frac{\mathit{mtype}(m, C.R) \text{ is undefined}}{\mathit{mtype}(m, C.R :: T) = \mathit{mtype}(m, T)}$$

Figure 11: Method type lookup

$$\frac{\texttt{class}\ C\ \{\ \bar{T}\ \bar{f};\ \bar{M}\ \bar{A}\ \} \quad T\ m(\bar{S}\ \bar{x})\{\ \texttt{return}\ e;\ \} \in \bar{M}}{\mathit{mbody}(m, \texttt{new}\ C(\bar{e})) = \bar{x}.e}\ \text{(MB-CLASS)}$$

$$\frac{\texttt{class}\ C\ \{\ \cdots \bar{A}\ \} \quad \texttt{role}\ R\ \texttt{requires}\ \{\ \bar{M}_I\ \}\{\ \cdots\ \bar{M}\ \} \in \bar{A} \quad T\ m(\bar{S}\ \bar{x})\{\ \texttt{return}\ e;\ \} \in \bar{M}}{\mathit{mbody}(m, \texttt{new}\ C(\bar{e}).R(\bar{d})\bar{r}) = \bar{x}.e, \texttt{new}\ C(\bar{e}).R(\bar{d})}\ \text{(MB-ROLE)}$$

$$\frac{\texttt{class}\ C\ \{\ \cdots \bar{A}\ \} \qquad m \notin \bar{M} \quad \texttt{role}\ R\ \texttt{requires}\ \{\ \bar{M}_I\ \}\{\ \cdots\ \bar{M}\ \} \in \bar{A}}{\mathit{mbody}(m, \texttt{new}\ C(\bar{e}).R(\bar{d})\bar{r}) = \mathit{mbody}(m, \bar{r})}\ \text{(MB-MIXIN)}$$

Figure 12: Method body lookup

Method lookup functions are defined in Figures 11 and 12. The type of method invocation m at T_S, written $\mathit{mtype}(m, T_S)$, is a pair of a sequence $\bar{T}$ of the formal parameter types and a return type T, written as $\bar{T} \to T$. We write $m \notin \bar{M}$ to imply that the method definition of the name m is not included in $\bar{M}$. The definition is also quite similar to field lookup functions; if T_S matches a mixin composition type $C.R :: T$, method declarations on $C.R$ are searched first. The body of the method invocation m on the sequence of role instances $\bar{r}$, written as $\mathit{mbody}(m, \bar{r})$, is a triple, written as $\bar{x}.e, r$, of the sequence of parameters $\bar{x}$, body e, and role instance r indicating where the method m is found from $\bar{r}$. The body

Bindability checking:

$$\frac{C \texttt{<:} \{\bar{M}_I\} \qquad \texttt{class}\ D\ \{\cdots\ \bar{A}\} \qquad \texttt{role}\ R\ \texttt{requires}\ \{\bar{M}_I\}\{\ \cdots\ \} \in \bar{A}}{\mathit{bindable}(D.R, C)}$$

Allowed unbinding:

$$\frac{T = \bar{D}.\bar{Q} :: C \qquad \forall D_i.Q_i \in \bar{D}.\bar{Q}, D_i.Q_i \in \bar{C}.\bar{Q}}{\mathit{unbindAllowed}(T, \bar{C}.\bar{Q})}$$

Figure 13: Other auxiliary functions

Well-formed role instance:

$$\frac{\mathit{bindable}(D.R, C) \qquad \mathit{fields}(D.R) = \bar{T}\ \bar{f} \qquad \Gamma \vdash \bar{e} : \bar{U} \qquad \bar{U} \texttt{<:} \bar{T}}{\Gamma \vdash \mathit{roleOK}(D, R, \bar{e}, C)}$$

Expression typing:

$$\Gamma \vdash x : \Gamma(x) \qquad \text{(T-VAR)}$$

$$\frac{\Gamma \vdash e_0 : S \qquad \mathit{ftype}(f, S) = T}{\Gamma \vdash e_0.f : T} \qquad \text{(T-FIELD)}$$

$$\frac{\Gamma \vdash e_0 : T_S \qquad \Gamma \vdash \bar{e} : \bar{S} \qquad \mathit{mtype}(m, T_S) = \bar{T} \to T \qquad \bar{S} \texttt{<:} \bar{T}}{\Gamma \vdash e_0.m(\bar{e}) : T} \qquad \text{(T-INVK)}$$

$$\frac{\mathit{fields}(C) = \bar{T}\ \bar{f} \qquad \Gamma \vdash \bar{e} : \bar{S} \qquad \bar{S} \texttt{<:} \bar{T} \qquad r_i = d_i.R_i(\bar{c}_i) \qquad \Gamma \vdash d_i : U_i \qquad U_i \texttt{<:} C_i \qquad \Gamma \vdash \mathit{roleOK}(C_i, R_i, \bar{c}_i, C)}{\Gamma \vdash \texttt{new}\ C(\bar{e}) \oplus \bar{r} : \bar{C}.\bar{R} :: C} \qquad \text{(T-NEW)}$$

$$\frac{\Gamma(\bar{x} : \bar{C}.\bar{R} :: \Gamma(\bar{x}), \bar{y} : \Gamma(\bar{y})/\bar{C}.\bar{R}) \vdash e_0 : T \qquad r_i = d_i.R_i(\bar{c}_i) \qquad \Gamma \vdash \bar{x} : \bar{S} \qquad \Gamma \vdash \bar{d} : \bar{U} \qquad \bar{U} \texttt{<:} \bar{C} \qquad \Gamma \vdash \mathit{roleOK}(C_i, R_i, \bar{c}_i, S_i) \qquad \Gamma \vdash \bar{y} : \bar{V} \qquad \Gamma \vdash \mathit{unbindAllowed}(V_i, \bar{C}.\bar{R})}{\Gamma \vdash \texttt{bind}\ \bar{x}\ \texttt{with}\ \bar{r}\ \texttt{from}\ \bar{y}\ \{\ \bar{x}\bar{y}.e_0\ \} : T} \qquad \text{(T-BIND)}$$

Figure 14: Expression typing

of the method invocation m on a class instance $\texttt{new}\ C(\bar{e})$, written as $\mathit{mbody}(m, \texttt{new}\ C(\bar{e}))$, is also defined in a similar manner.

Other auxiliary definitions regarding binding operations are shown in Figure 13. The predicate $\mathit{bindable}(D.R, C)$ checks whether or not an instance of C can be bound with an instance of $D.R$. This predicate returns true if C is a subtype of $D.R$'s required interface $\{\ \bar{M}_I\ \}$. Finally, the predicate $\mathit{unbindAllowed}(T, \bar{C}.\bar{R})$ checks whether all the role types contained in T are also members of $\bar{C}.\bar{Q}$.

3.3 Typing

$$\frac{\bar{x} : \bar{T}, \texttt{this} : C \vdash e_0 : T_0 \qquad \texttt{class}\ C\ \{\ \cdots\ \} \qquad \bar{C} \in \mathit{dom}(CT) \qquad \bar{R}\ \texttt{OK IN}\ \bar{C}}{T_0\ m(\bar{T}\ \bar{x})\ \{\ \texttt{return}\ e_0;\ \}\ \texttt{OK IN}\ C} \qquad \text{(T-METHOD)}$$

$$\frac{\bar{x} : \bar{T}, \texttt{thisC} : C, \texttt{this} : C.R, \texttt{super} : \{\ \bar{M}_I\ \} \vdash e_0 : T_0 \qquad \texttt{class}\ C\ \{\ \cdots\ \bar{A}\ \} \qquad \bar{C} \in \mathit{dom}(CT) \qquad \bar{R}\ \texttt{OK IN}\ \bar{C} \qquad \texttt{role}\ R\ \{\ \bar{M}_I;\ \}\{\ \cdots\ \} \in \bar{A}}{T_0\ m(\bar{T}\ \bar{x})\ \{\ \texttt{return}\ e_0;\ \}\ \texttt{OK IN}\ C.R} \qquad \text{(T-RMETHOD)}$$

$$\frac{\bar{M}\ \texttt{OK IN}\ C.R}{\texttt{role}\ R\ \texttt{requires}\ \{\ \bar{M}_I\ \}\{\ \bar{T}\ \bar{f};\ \bar{M}\ \}\ \texttt{OK IN}\ C} \qquad \text{(T-ROLE)}$$

$$\frac{\bar{M}\ \texttt{OK IN}\ C \qquad \bar{A}\ \texttt{OK IN}\ C}{\texttt{class}\ C\ \{\ \bar{T}\ \bar{f};\ \bar{M}\ \bar{A}\ \}\ \texttt{OK}} \qquad \text{(T-CLASS)}$$

Figure 15: Well-formed definitions

The typing rules for FEJ expressions are shown in Figure 14. An environment Γ is a finite mapping from variables to types, written as $\bar{x} : \bar{T}$. The typing judgment for expressions has the form $\Gamma \vdash e : T$, read as "in the environment Γ, expression e has type T."

The rules are syntax directed, with one rule for each form of expressions. The typing rules for method invocations and class instance creations check whether each actual parameter has a type of the corresponding formal parameter. The typing rule for class instance creations also checks that each role instance bound with the class instance is well-formed (i.e., each actual parameter for the role instance creation has a type of the corresponding formal parameter) and the class C is a subtype of each type of role instance.

The rule T-BIND is complicated. By $\Gamma(\bar{x} : \bar{T})$, we imply an environment that can be obtained by updating all the types of $\bar{x}$ contained in Γ to the corresponding types $\bar{T}$, respectively. By $T/C.R$, we imply a type generated by removing $C.R$ from T, if T is a type of the form $\bar{C}.\bar{R} :: C$ and $C.R$ is contained in $\bar{C}.\bar{R}$. We write $\bar{x} : \bar{C}.\bar{R} :: \Gamma(\bar{x})$ as a shorthand for $x_1 : C_1.R_1 :: \Gamma(x_1), \cdots, x_n : C_n.R_n :: \Gamma(x_n)$, and $\bar{y} : \Gamma(\bar{y})/\bar{C}.\bar{R}$ as a shorthand for $y_1 : \Gamma(y_1)/C_1.R_1, \cdots, y_n : \Gamma(y_n)/C_n.R_n$. Thus, the first hypothesis of T-BIND indicates that under the updated type environment, the body of `bind` expression e_0 has type T. Then, this rule inspects the role names of $\bar{r}$ and checks whether the types of $\bar{x}$ are compatible with those roles. It also checks whether role unbinding of $\bar{C}.\bar{R}$ is allowed for $\bar{y}$.

Typing rules for method declarations, role declarations, and class declarations are shown in Figure 15. The type of the body of a method declaration is a subtype of the return type. The special variable `this` is bound in every method declaration, and for every method declaration in roles, variables `thisC` and `super` are also bound; the type of `thisC` is its enclosing class, and the type of `super` is its required interface. A role declaration is well-formed if all the methods declared in that role are well-formed, and a class declaration is well-formed if all the methods and roles

$$\frac{\mathit{fvalue}(f, \texttt{new}\ C(\bar{v}) \oplus \bar{r}) = w}{(\texttt{new}\ C(\bar{v}) \oplus \bar{r}).f \longrightarrow w} \quad \text{(R-FIELD)}$$

$$\frac{v = \texttt{new}\ C(\bar{v}') \oplus \bar{r} \qquad \mathit{mbody}(m, \bar{r}) \text{ is undefined} \qquad \mathit{mbody}(m, \texttt{new}\ C(\bar{v}')) = x.e}{v.m(\bar{v}) \longrightarrow [\bar{v}/\bar{x}, v/\texttt{this}]e} \quad \text{(R-INVK)}$$

$$\frac{v = \texttt{new}\ C(\bar{v}') \oplus \bar{r} \qquad \bar{r} = \bar{r}_1, w.R(\bar{w}), \bar{r}_2 \qquad \mathit{mbody}(m, \bar{r}) = \bar{x}.e, w.R(\bar{w}) \qquad \mathit{cp}(v) = \texttt{new}\ C(\bar{v}') \oplus \bar{r}_2}{v.m(\bar{v}) \longrightarrow [\bar{v}/\bar{x}, v/\texttt{this}, w/\texttt{thisC}, \mathit{cp}(v)/\texttt{super}]e} \quad \text{(R-RINVK)}$$

$$\texttt{bind}\ \bar{v}\ \texttt{with}\ \bar{r}\ \texttt{from}\ \bar{w}\ \{\bar{x}\bar{y}.e_0\} \longrightarrow [(\bar{v} \oplus \bar{r})/\bar{x}, (\bar{w} - \bar{r})/\bar{y}]e_0 \quad \text{(R-BIND)}$$

$$\frac{e \longrightarrow e'}{e.f \longrightarrow e'.f} \quad \text{(CR-FIELD)}$$

$$\frac{e_0 \longrightarrow e_0'}{e_0.m(\bar{e}) \longrightarrow e_0'.m(\bar{e})} \quad \text{(CR-INVK)}$$

$$\frac{e_i \longrightarrow e_i'}{v.m(\cdots, e_i, \cdots) \longrightarrow v.m(\cdots, e_i', \cdots)} \quad \text{(CR-INVK-ARG)}$$

$$\frac{e_i \longrightarrow e_i'}{\texttt{new}\ C(\cdots, e_i, \cdots) \longrightarrow \texttt{new}\ C(\cdots, e_i', \cdots)} \quad \text{(CR-NEW)}$$

Figure 16: Dynamic semantics

declared in that class are well-formed.

3.4 Dynamic semantics

The reduction rules of FEJ are shown in Figure 16. We use a standard call-by-value operational semantics. There exist four computation rules: R-FIELD, R-INVK, R-RINVK, and R-BIND. The rest of the rules formalize the call-by-value strategy. The reduction relation is of the form $e \longrightarrow e'$, read "expression e reduces to expression e' in one step."

For the rule R-FIELD, the field f is searched in all the role instances $\bar{r}$ bound with the class instance $\texttt{new}\ C(\bar{v})$. There exist two rules for method invocation: one is for method invocation declared in a class instance (rule R-INVK), and the other is for method invocation declared in a role instance (rule R-RINVK). The method invocation reduces to the expression of the method body, substituting all the formal parameters $\bar{x}$ with the argument values $\bar{v}$ and the special variable `this` with the receiver of method invocation. Furthermore, in R-RINVK, the special variables `thisC` and `super` are also substituted with corresponding values; `thisC` is substituted with the enclosing class instance returned by *mbody*. By $\mathit{cp}(v)$, we imply a fresh value whose class name and arguments for its instance creation are identical to v. In R-RINVK, a new value copied from the receiver of method invocation is created and replaced with `super`. The binding role instances for $\mathit{cp}(v)$ are also changed so that $\mathit{cp}(v)$ is bound with role instances that exist on right-hand side of $w.R(\bar{w})$ (role instance returned by *mbody*), which formalizes the method combination mechanism for role instance compositions.

The `bind` expression reduces to its body. It substitutes the free variables $\bar{x}$ and $\bar{y}$ with values appearing in the `bind` clause and `from` clause, respectively. By $v \oplus r$, we imply a value obtained by adding r to the left-most side of v's binding roles $\bar{r}$. By $v - r$, we imply a value obtained by removing r from the v's binding roles. We write $(\bar{v} \oplus \bar{r})/\bar{x}$ as a shorthand of $(v_1 \oplus r_1)/x_1, \cdots, (v_n \oplus r_n)/x_n$. Similarly, we write $(\bar{w} - \bar{r})/\bar{x}$ as a shorthand of $(v_1 - r_1)/x_1, \cdots, (v_n - r_n)/x_n$. Thus, in the resulting expression, roles $\bar{r}$ are removed from $\bar{w}$ and added to $\bar{v}$. Because FEJ is a purely functional calculus, both binding operations (creating a new role instance) and context activation (activating the role instance) are expressed in one reduction rule.

3.5 Property

We show the property of FEJ, namely, every well-typed expression evaluates to a value. In this section, we only present a series of theorems indicating FEJ type soundness. We provide proofs in the full version of this paper[5].

THEOREM 3.1 (SUBJECT REDUCTION). *If* $\Gamma \vdash e : T$ *and* $e \longrightarrow e'$, *then* $\Gamma \vdash e' : S$ *for some* $S <: T$.

THEOREM 3.2 (PROGRESS). *If* $\Gamma \vdash e : T$ *and there exist no* e' *such that* $e \longrightarrow e'$, *then* e *is a value.*

THEOREM 3.3 (FEJ TYPE SOUNDNESS). *If* $\emptyset \vdash e : T$ *and* $e \longrightarrow^* e'$ *with* e' *a normal form, then* e' *is a value* v *with* $\emptyset \vdash v : S$ *and* $S <: T$.

4. RELATED WORK

We have overviewed the main features of NextEJ and compare it with those of EpsilonJ. Besides the object adaptation mechanism of EpsilonJ, NextEJ provides the feature of context activation inspired by COP languages. However, context activation in NextEJ is slightly different from that of COP languages. While COP languages provide methods for realizing behavioral variability with respect to some contextual information, NextEJ puts more emphasis on acquiring and activating *new* behaviors that are not provided by the original class.

The Epsilon model, on which this work is based, is related to aspect-oriented programming (AOP). AOP has a useful feature in that it enables one to add aspects dynamically as well as statically [29]. One of the main AOP languages is AspectJ [30], a Java-based AOP language. The main objective of writing aspects is to deal with cross-cutting concerns. This implies that there already exists some structure of module decomposition. Although efforts have been made to design software based on the AOP principle from the beginning, the normal framework of mind for thinking aspects assumes the existing program code as a target for inserting advices to join points. Instead, Epsilon does not assume any existing code and designs collaboration contexts independently. The work corresponding to designating pointcuts and attaching advices is executed by binding objects to roles.

[5] `http://www.l.u-tokyo.ac.jp/~kamina/nej-full.pdf`

The Epsilon model is also related to feature-oriented programming (FOP) in that both approaches provide a method to modularize and compose features. FeatureC++ is an FOP extension for the C++ programming language that provides a method for composing features statically as well as dynamically [3, 39]. In FeatureC++, a developer can select a composition method from static binding or dynamic binding when composing a product. Context-activation mechanism is not considered in FeatureC++.

In CaesarJ, a Java-based AOP language, an aspect can be deployed and undeployed at any time [4]. This feature is similar to context activation scope in NextEJ. However, in NextEJ, a context is activated at the binding time. On the other hand, in CaesarJ, the binding is specified in the *binding class*. Although CaesarJ's binding class and methods for aspect deployment provide much flexibility for aspect composition and activation, NextEJ provides a more simple and flexible basis for object adaptation and context activation mechanism by specifying objects and contexts at the binding time.

ObjectTeams [21] also has a similar mechanism for role binding. In ObjectTeams, each instance of a bound role class internally stores a reference to its base object. This reference cannot be changed during its lifetime. By *lowering* (retrieving the base object from a role object) and *lifting* (opposite of lowering), we can safely change the behavior of the object at run-time. As in NextEJ, a *team* (a construct of ObjectTeams corresponds to a context in NextEJ) can be activated and deactivated. However, in ObjectTeams, the role-class binding is declared at the class declaring time. NextEJ and EpsilonJ provide a more flexible method to express the relationship between roles and classes by using `requires` clause. Currently there exist no formalizations of ObjectTeams.

Delegation Layers [37] provide flexible object-based composition of collaborations. They combine the mechanism of delegation [31, 38] and virtual classes [32, 10], or Family Polymorphism [15]; roles may be represented by virtual classes, and a composition is instance-based using the delegation mechanism. However, this approach do not successfully represent the object adaptation described in this paper. For example, in NextEJ, the object can assume and discard a role dynamically, and even the discarded role may be assumed by another object and the state held in the role instance is taken over by the latter object.

powerJava [5] is a language similar to NextEJ in that roles and collaboration fields are the first class constructs, interaction between roles are encapsulated, and objects can participate in the interaction by assuming one of its roles. As in NextEJ, the type of role depends on the enclosing context instance. However, powerJava lacks the feature of role groups, a powerful mechanism for obtaining role instances associated with the context instance reflectively. Role unbinding and swapping, and explicit ordering of context activation that affects method combination are also unconsidered.

There are pieces of literature that formalize the feature of extending objects at run-time. Ghelli presented foundations for extensible objects with roles based on Abadi-Cardelli's object calculi [1], where coexistence of different methods introduced by incompatible extensions is considered [19]. Gianantonio et al. presented a calculus $\lambda Obj+$[20], an extension of λObj[17] with a type assignment system that allows a self-inflicted object extension that still statically catches the "message not found" errors. Drossopoulou et al. proposed a type-safe core language $Fickle$[14] that allows reclassification of objects, a mechanism for dynamically changing object's belonging classes that share the same "root" superclass. On the other hand, FEJ focuses on a foundation of object adaptation and context activation for Java-like languages (based on FJ). FEJ supports a notable feature of unbinding of role instances (removing role instances from values in `from` clause of `bind` expression).

Mixins [9] are related to roles in NextEJ in that they form partial definitions that can be reused with a number of classes that conform to the requirements of mixins. Several extensions of Java with mixins have been proposed [18, 2, 26]. Although mixin composition is originally performed at compile time, dynamic composition of mixins is also studied in a core calculus [7], and such kind of object-level inheritance is also studied as *wrappers* [11, 8].

5. CONCLUDING REMARKS

We have presented NextEJ, a type-safe alternative to Epsilon model programming language with the features of COP. It provides a method for naturally representing context-awareness at the programming language level. Based on the object adaptation mechanism provided by Epsilon model, NextEJ supports a convenient COP feature of activating/deactivating contexts or roles. While such activation is only supported by down-casting in EpsilonJ, NextEJ provides a safe way for activation by the context activation scope. Furthermore, in NextEJ, multiple contexts can be activated at a time, and the behavior of the composite context generated by such multiple context activations is determined by the order of activations. A small core calculus FEJ provides a solid basis for assuring its type soundness.

Acknowledgments.

This work is supported in part by a Grant-in-Aid for Scientific Research No.18200001 and Grant-in-Aid for Young Scientists (B) No.20700022 from NEXT of Japan.

6. REFERENCES

[1] Martin Abadi and Luca Cardelli. *A Theory of Objects*. Springer, 1996.

[2] Davide Ancona, Giovanni Lagorio, and Elena Zucca. Jam – designing a Java extension with mixins. *ACM TOPLAS*, 25(5):641–712, 2003.

[3] S. Apel, T. Leich, and G. Saake. FeatureC++: On the symbiosis of feature-oriented and aspect-oriented programming. In *GPCE'05*, volume 3676 of *LNCS*, pages 125–140, 2005.

[4] 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.

[5] M. Baldoni, G. Boella, and L. van der Torre. Interaction between objects in powerJava. *Journal of Object Technology*, 6(2):5–30, 2007.

[6] Masahiro Bessho, Shinsuke Kobayashi, Noboru Koshizuka, and Ken Sakamura. A space-identifying ubiquitous infrastructure and its application for tour-guiding services. In *SAC 2008*, pages 1516–1621, 2009.

[7] Lorenzo Bettini, Viviana Bono, and Silvia Likavec. Safe and flexible objects with subtyping. *Journal of Object Technology*, 4(10):5–29, 2005.

[8] Lorenzo Bettini, Sara Capecchi, and Elena Giachino. Weatherweight Wrap Java. In *SAC'07*, pages 1094–1100, 2007.

[9] G. Bracha and W. Cook. Mixin-based inheritance. In *OOPSLA 1990*, pages 303–311, 1990.

[10] Kim B. Bruce, Martin Odersky, and Philip Wadler. A statically safe alternative to virtual types. In *ECOOP'98*, volume 1445 of *LNCS*, pages 523–549, 1998.

[11] Martin Buchi and Wolfgang Weck. Generic wrappers. In *ECOOP 2000*, volume 1850 of *LNCS*, pages 201–225, 2000.

[12] Dave Clarke and Ilya Sergey. A semantics for context-oriented programming with layers. In *COP'09*, page No.10, 2009.

[13] 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.

[14] Sophia Drossopoulou, Ferruccio Damiani, and Mariangiola Dezani-Ciancaglini. Fickle: Dynamic object re-classification. In *ECOOP 2001*, volume 2072 of *LNCS*, pages 130–149, 2001.

[15] Eric Ernst. Family polymorphism. In *ECOOP 2001*, volume 2072 of *LNCS*, pages 303–327, 2001.

[16] Erik Ernst. Propagating class and method combination. In *ECOOP'99*, volume 1628 of *LNCS*, pages 67–91. Springer-Verlag, 1999.

[17] K. Fisher, F. Honsell, and J.C̃. Mitchell. A lambda calculus of objects and method specialization. *Nordic Journal of Computing*, 1(1):3–37, 1994.

[18] Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. Classes and mixins. In *POPL 98*, pages 171–183, 1998.

[19] Giorgio Ghelli. Foundations for extensible objects with roles. *Information and Computation*, (175):50–75, 2002.

[20] Pietro Di Gianantonio, Furio Honsell, and Luigi Liquori. A lambda calculus of objects with self-inflicted extension. In *OOPSLA'98*, pages 166–178, 1998.

[21] Stephan Herrmann. A precise model for contextual roles: The programming language ObjectTeams/Java. *Applied Ontology*, 2(2):181–207, 2007.

[22] 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.

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

[24] Atsushi Igarashi, Benjamin Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM TOPLAS*, 23(3):396–450, 2001.

[25] Atsushi Igarashi and Mirko Viroli. Variant path types for scalable extensibility. In *OOPSLA'07*, pages 113–132, 2007.

[26] Tetsuo Kamina and Tetsuo Tamai. McJava – a design and implementation of Java with mixin-types. In *2nd ASIAN Symposium on Programming Languages and Systems (APLAS 2004)*, volume 3302 of *LNCS*, pages 398–414. Springer, 2004.

[27] Tetsuo Kamina and Tetsuo Tamai. Flexible object adaptation for Java-like languages. In *Proceedings of the 10th Workshop on Formal Techniques for Java-like Programs (FTfJP 2008)*, pages 63–76, 2008.

[28] Tetsuo Kamina and Tetsuo Tamai. Towards safe and flexible object adaptation. In *COP'09*, page No.4, 2009.

[29] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *ECOOP'97*, 1997.

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

[31] Gunter Kniesel. Type-safe delegation for run-time component adaptation. In *ECOOP'99*, volume 1628 of *LNCS*, pages 351–366, 1999.

[32] Ole Lehrmann Madsen and Birger Møller-Pedersen. Virtual classes: A powerful mechanism in object-oriented programming. In *OOPSLA'89*, pages 397–406, 1989.

[33] Ole Lehrmann Madsen, Birger Møller-Pedersen, and Kristen Nygaard. *Object-Oriented Programming in the BETA Programming Language*. Addison-Wesley, 1993.

[34] Supasit Monpratarnchai and Tetsuo Tamai. The implementation and execution framework of a role model based language, EpsilonJ. In *SNPD'08*, pages 269–276, 2008.

[35] Nathaniel Nystrom, Stephen Chong, and Andrew C. Myers. Scalable extensibility via nested inheritance. In *OOPSLA'04*, pages 99–115, 2004.

[36] Nathaniel Nystrom, Xin Qi, and Andrew C. Myers. *J&*: Nested intersection for scalable composition. In *OOPSLA'06*, pages 21–35, 2006.

[37] Klaus Ostermann. Dynamically composable collaborations with delegation layers. In *ECOOP 2002*, volume 2374 of *LNCS*, pages 89–110, 2002.

[38] Klaus Ostermann and Mira Mezini. Object-oriented composition untangled. In *OOPSLA'01*, pages 283–299, 2001.

[39] Marko Rosenmüller and Norbert Siegmund. Code generation to support static and dynamic composition of software product lines. In *GPCE'08*, pages 3–12, 2008.

[40] Chieri Saito, Atsushi Igarashi, and Mirko Viroli. Lightweight family polymorphism. *Journal of Functional Programming*, 18(3):285–331, 2008.

[41] Tetsuo Tamai, Naoyasu Ubayashi, and Ryoichi Ichiyama. An adaptive object model with dynamic role binding. In *International Conference on Software Engineering (ICSE 2005)*, pages 166–175, 2005.

[42] Tetsuo Tamai, Naoyasu Ubayashi, and Ryoichi Ichiyama. Objects as actors assuming roles in the environment. In *Software Engineering for Multi-Agent Systems V*, volume 4408 of *LNCS*, pages 185–203, 2007.

Towards An Open Trace-Based Mechanism

position paper

Paul Leger Éric Tanter
PLEIAD Laboratory
Computer Science Department (DCC)
University of Chile – Santiago, Chile
http://www.pleiad.cl

ABSTRACT

Real-world applications have to deal with issues related to security, as well as errors and crosscutting concerns. Different trace-based mechanisms with distinctive features have been proposed to solve these particular issues. For example, PQL matches sequence of unordered events, and tracematches match traces expressed with regular expressions. Despite that applications present these issues at the same time, there is not a single trace-based mechanism that supports the distinctive features of current mechanisms. Besides, lack of an expressive trace-based mechanism does not permit to include new features, therefore, developers end up "code around" these mechanisms to satisfy particular needs. In this position paper, we compare and relate the specific characteristics of current trace-based mechanisms. Finally, we present a model for an open trace-based mechanism.

1. INTRODUCTION

Nowadays, real-world applications have to deal with issues related to security, as well as errors and crosscutting concerns. *Trace-based Mechanisms* (TMs for short) have shown their usefulness to solve some of these issues [1, 3, 5, 8, 10]. A TM observes the execution of the software at runtime and (possibly) executes a code fragment when the TM matches a specified trace of the execution. The researchers have proposed specific TMs to resolve these particular issues. Unsurprisingly, applications present these issues at the same time, therefore, they need to use different TMs at the same time, because there is not a single TM that supports the distinctive of current mechanisms. Besides, lack of an expressive TM does not allow developers to include new features, therefore, they end up "code around" these mechanism in contort ways. In this position paper, we first describe an abstract operational model of TMs (Section 2), which we then use to relate and compare the specific characteristics of existing TMs (Section 3). Section 4 describes the design of an open TM based on this abstract model. The open model is formulated in a class-based object-oriented setting and follows the design guidelines of open implementations [7]. We illustrate the range of openness of the model by describing several concrete extensions. For example:

- Express sequences using *operators*, which can enjoy all the power of the base language to be defined.
- Manage expressively the granularity of the matching, from a particular sequence to all sequences of a TM.
- Control expressively the multiplication of sequences of a TM.

2. AN OVERVIEW OF TMS

In this section, we describe an abstract operational model of TMs. We divide this abstract model in two parts: first, we describe the abstract model necessary to match a specified trace of execution. Later, we extend this abstract model to manage and control the matching of several traces of executions at the same time.

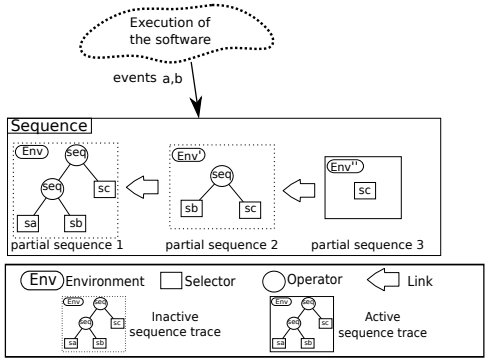


Figure 1: A sequence in action.

2.1 Matching a Trace

Figure 1 shows that the execution of the software has generated the trace of events *a b* and that the *sequence* has matched these two events. According to the figure, the sequence needs to match the *c* event to finish the matching. When the sequence finishes the matching, our model executes an extra piece of code similar to an advice in AspectJ [6]. For space reasons, we only focus on the matching of sequences in this paper.

A sequence matches a trace of execution, which in this case is the trace composed of events: *a b c*. In addition, a sequence contains a set of linked *partial sequences*. A partial sequence represents the trace of execution that the sequence need to match. The first partial sequence represents the *whole* sequence and the last represents the last event that the sequence needs to match. Only active partial sequences are only evaluated and so can match events. The set of partial sequences represents the history of the matching of a sequence, which we then use to support the multiple matching of sequences. A partial sequence is composed of an Abstract Sequence Tree (AST), like AST for source code of a programming language, and an *environment* of bindings. In this AST, *selectors* are the terms and *operators* are combinators. Selectors match single events of the trace of execution and operators combine partial sequences. The environment contains a set of bindings available for a partial sequence and for the execution of the extra piece of code when the sequence matches. Every time that a partial sequence matches an event, the root operator of the AST creates a new par-

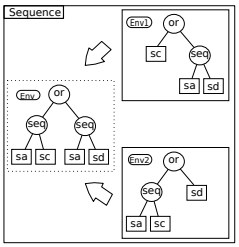


Figure 2: The non-deterministic effect of the Or operator.

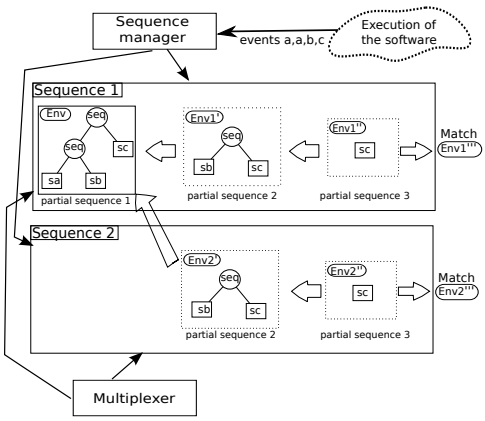


Figure 3: A TM in action.

tial sequence that is based and linked with the previous one. This new partial sequence contains a modified environment and represents the remaining sub-sequence of the sequence. A sequence can activate and deactivate its partial sequences. For example, when the partial sequence 2 matched the *b* event, the partial sequence 3 was created; and the sequence activated the partial sequence 3 and deactivated the partial sequence 2. Although the *whole* sequence did not match with the *b* event, this sequence advanced because now this sequence only has to match the *c* event to match entirely.

Operators can create one or more partial sequence when an event is matched. For example, Figure 2 shows a single sequence that has a partial sequence with the or operator over two sequences: *a c* and *a d*. When the *a* event occurs, two different histories can happen: the first or second sequence advance in the matching. It is so because the or operator is non-deterministic and so produces two different histories of the matching of the same sequence.

2.2 Matching Several Traces

Now, we extend this abstract model to manage and control the matching of several sequences inside a TM. Figure 3 shows a new scenario and two new components of our model: *sequence manager* and *multiplexer*. In the figure, the trace of execution has generated the events *a a b c* and the sequence manager has received and sent these events to the two sequences, which has matched.

The sequence manager manages the matching of all sequences inside a TM. For instance, this components permits to abort or reset the matching of all or a specific group sequences. In this case, the sequence manager only sends the events to sequences.

The multiplexer controls the multiplication of sequences. When a sequence generate an *equivalent* partial sequence the multiplexer decides if this equivalent partial sequence becomes in a new sequence. For example, Figure 3 shows that a sequence is multiplexed into two, which end up matching. This double matching is due to four reasons: *i)* the trace of execution generates twice the *a* event, *ii)* the sequence 1 always has the partial sequence 1 activated, *iii)* the multiplexer decides to create a new sequence when the sequence 1 matches the second *a* event and creates an equivalent partial sequence 2, *iv)* and when the trace of execution generates finally the events *b* and *c*, sequences 1 and 2 advance up to the matching. In other words, when the execution of the software generates the second *a* event, the partial sequence 1, which is activate, matches the event, therefore, creates a new partial sequence 2. As there are two equivalent partial sequences, the multiplexer decides to create another sequence with the second partial sequence 2.

It is important to note that the nondeterminism differs from the multiplication of sequences because the first generates different histories of the same sequence using the same event; instead the second generates different histories in different sequences using different events.

In this section, we appreciate six components of TMs: environments, selectors, operators, sequences, sequence managers, and multiplexers. In next section, we will compare and relate the expressiveness of these components in existing TMs.

3. THE EXPRESSIVENESS OF TMS

TMs [1, 3, 5, 8, 10] vary in their expressiveness in terms of environments, selectors, operators, sequences, sequence managers, and multiplexers. In this section, we relate and compare the expressiveness of these components.

Environment. An environment of bindings represents the contextual information associated to the sequence. The expressiveness of environments allows developers to match more precisely a trace of execution and to provide more contextual information (values) to the execution of the code fragment when the sequence matches. In tracematches [1], the manipulation of environments is limited because it only permits to bind information related to the event and compare implicitily this information using the equality operator. Environments of Alpha [10] can only contain information related to events. Halo [5] allows developers to contain contextual information from any source (not only from the event), and then comparing this information explicitly.

Selector. A selector matches single events. The precision to match events depends on the granularity of the event model of the base language and the expressiveness to define selectors. Although this expressiveness varies, most of current TMs [1, 3, 5, 8, 10] cannot use the power of the base language to define selectors. For instance, in tracematches and Halo, selectors are pointcuts defined in a dedicated declarative language[1]. Selectors of Alpha are Prolog queries, which differs from the base language (a Java subset). In PTQL [3], selectors are fields of a register of a data base.

Operator. An operator relates selectors and/or operators, therefore, it permits to define partial sequences. The expressiveness to define operators varies according to TMs. For example, the expressiveness of selectors of Alpha is enough to express operators because it can express sequences using Prolog queries[2]. In Halo, operators

[1]In tracematches, a selector is really a symbol that is composed of a pointcut and a modifier of the event.

[2]To be more precise, the selectors are facts and the operators are rules.

are Lisp functions defined by the def-rule construct. The operators of PTQL are SQL operators like or and and. Tracematches match traces of execution using regular expression operators. For example, if the alphabet is *{sa,sb}* and the regular expression of the sequence is *sa sa sa* and the regular expression of the trace of execution is *sa sa sb sa*, so tracematches do not match this trace because the *sb* symbol of the execution is not in the regular expression of the sequence. In a nutshell, the regular expression of the trace of execution must happen exactly as the sequence is defined.

Sequence. Although an environment and an AST of the sequence define a partial sequence, they do not define entirely the process of matching of a sequence. A sequence handles the set of partial sequences to carry out the matching and the history of this matching. Sadly, to the best of our knowledge, there is no TM that allows developers to reason or reflect about of the matching of a particular sequence. Reasoning about a sequence permits, for example, to abort or reset the matching of a particular sequence if some condition is satisfied. Concretely, consider the familiar example for AOP community: *autosave*. A document is automatically saved if it is edited a number of times (*e.g.* say three) without being saved. The wanted sequence should match when the trace of execution generates three events of edition, but this sequence should abort if it matches the sequence composed of the *save* event because the document has already saved.

Sequence manager. The sequence manager manages the matching of all sequences inside a TM. To the best of our knowledge, there is no TM that permits to manage this component. Reason about the matching of sequences permits, for example, to abort or reset the matching of all sequences if some condition is satisfied. Concretely, controlling the sequence manager can be useful in areas like security. For instance, a sequence that represents a protocol of *light security* could change to *heavy security* whenever another sequence matches. This example shows that controlling the sequence manager permits to obtain behavior similar to *morphing aspects* [4] in TMs.

Multiplexer. The multiplexer controls the multiple matching of sequences. In most of these mechanisms [3, 5, 8, 10], the sequence are always multiplexed. Tracematches are a particular case because a sequence is only duplicated when two equivalent partial sequences have environments of bindings with different values. The multiple matching is used, for example, in tracematches resolve to the problem of the observer pattern [2] because this TM matches multiple sequences that binds different values in the *subject* and *observer*.

4. A MODEL FOR AN OPEN TM

This section presents the design of an open TM, which is based on the abstract operational model presented in Section 2. The model follows the design guidelines of the open implementations [7] because this model allows developers to control its implementation strategy by the an expressive use of environments, selectors, operators, sequences, sequence managers, and multiplexers. In similar way to Section 2, this model is split into two: *partial sequence* and *sequence manager*. The first model permits define sequences by the use of environments, selectors, and operators. The second model permits to describe sequence managers, which are used to manage and control the multiple matching of the sequences.

4.1 Defining a Partial Sequence

As mentioned in Section 2.1, a partial sequence represents the trace of execution that the sequence need to match. A partial sequence is a structure composed of an environment and an AST of the sequence. This AST settles the relationships between selectors and operators. The environment of bindings defines the set of values available in a partial sequence. In this model, the expressiveness of selectors depends on the event model of the base language, but the expressiveness of the operators depends on the base language, which is generally Turing Complete.

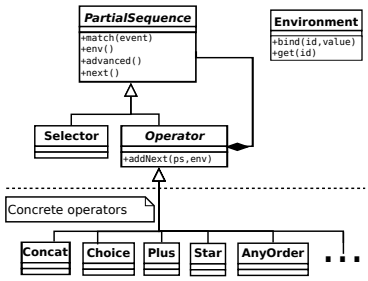


Figure 4: The class diagram of a partial sequence.

Figure 4 shows the class diagram of a partial sequence. This diagram uses the composite pattern [2] between PartialSequence, Selector, and Operator. The Selector class represents selectors that only match single events. The Operator class represents operators that match compositions of partial sequences. This class is an abstract class, which is used to implement sub-classes that provide specific and diverse kinds of operators. For example, Figure 4 shows five operators: four to match regular expressions and one to match traces in any order. The Environment class represents environments that permit to bind and get values.

The match method takes an event and returns true or false whether it matches or not a single event (in the selector case) or a partial sequence (in the operator case). The env method returns the environment that contains the bound values until this point of the sequence. If a partial sequence does not match but advance (like Figure 1), the advanced method returns true. The next method returns the next partial sequence. Finally, the addNext method of the Operator class adds the next partial sequence (ps) with the associated environment (env). This latter method determines which is the next partial sequence in the process of matching of a sequence.

The protocol of use is the following: when a PartialSequence object sets the next partial sequence with itself, we will say the matching of the sequence did not advance. Instead, when a PartialSequence object sets or adds the next partial sequences with different objects, we will say the matching of the sequence advanced. As examples, we present the implementation of the Concat and AnyOrder classes.

Figure 5 shows the Concat class, which represents the operator that matches the sequence of two traces, where both traces are matched by the left and right partial sequences. The match method adds right as the next partial sequence if left matches with the event. However, if left does not match, it could have advanced in its matching. In this case, match adds as next a new Concat object that contains the left.next() as left and right as right. A new Concat object is created to maintain the history of the matching of the sequence through the two different partial sequences (before and after the event).

It is important to note, the match method of Concat class always returns false because the responsibility of the matching is delegated to the right partial sequence.

The following code is useful if we want to match the sequence

```
class Concat extends Operator {
  PartialSequence left,right;
  Seq(PartialSequence left,PartialSequence right) {...}

  boolean match(Event event) {
    Env env = env();
    left.setEnv(env);
    if (left.match(event)) { //left matched, continue with right
        addNext(right,left.env());
    }
    else if (left.advanced()) { //left only advanced
      addNext(new Seq(left.next(),right),left.next().env()));
    }
    else {
      addNext(this,env);
    }
    return false;
} };
```

Figure 5: The Concat class.

composed of *a b c* events, where events are calls to functions a, b, and c:
Concat concat = new Concat(new Concat(sa,sb),sc);
The sa, sb, and sc objects are selectors that match the calls to aforementioned functions. The concat object is a partial sequence that represents the matching of the wanted sequence.

The anyOrder operator matches an unordered sequences of traces, where these traces are matched by a set partial sequences. This operator is non-deterministic like the *or* operator (Section 2) because the two or more partial sequences can match or advance with the same event, therefore, generating different histories of the matching. Figure 6 shows the implementation of the AnyOrder class that represents the AnyOrder operator. The match method, first, verifies if there is only a partial sequence and tries to match this last partial sequence. The match method returns true if the ps matches, but returns false otherwise. However, if ps only advanced, the match method adds ps.next() as next, which has the responsibility to finish the matching of the unordered sequences of traces.

When there are two or more partial sequences in the pss array, the match method tries to match every partial sequence (ps) of the pss array. The method adds a new AnyOrder object with the same pss array without ps if it matched; instead match adds a new AnyOrder object with the same pss array, but exchanges ps for ps.next() if ps only advanced.

The following code is useful if we want to match the unordered sequence composed of the events *a*, *b*, and *c*:
AnyOrder anyOrder = new AnyOrder(new PartialSequence[]{sa,sb,sc});
The anyOrder object is a partial sequence that represents the matching of the disorderly aforementioned events.

4.2 Defining a Sequence Manager

The previous section showed how to build partial sequences. In this section, we describe our sequence manager model, which takes a partial sequence to manage and control the matching and the multiplication of sequences generated from this partial sequence.

Figure 7 shows the class diagram of our sequence manager. In this diagram, three classes are the core: Sequence, Multiplexer, and SequenceManager. The Sequence class represents a sequence, which contains a set of (active and inactive) partial sequences. The Multiplexer class controls the multiplication of sequences. Finally, the SequenceManager class manages the matching of sequences.

4.2.1 Sequence

A sequence can be declared using the following code:
Sequence s = new Sequence(ips,as);

```
class AnyOrder extends Operator {
  PartialSequence[] pss;
  AnyOrder(PartialSequence[] pss) {...}

  boolean match(Event event) {
    Env env = env();
    if (pss.length == 1) { //only one partial sequence remains
      PartialSequence ps = pss[0];
      ps.setEnv(env);
      if (ps.match(event)) { //last partial sequence matched
        setEnv(ps.env());
        return true;
      } else if (ps.advanced()) { //delegate the matching to ps.next()
        addNext(ps.next(),ps.next().env());
        return false;
    } }

    for(int i = 0; i < pss.length; ++i) {
      PartialSequence ps = pss[i];
      ps.setEnv(env);
      if (ps.match(event)) { //ps[i] matched
        addNext(new AnyOrder(pss.remove(i)),ps.env());
      } else if (ps.advanced()) { //ps[i] advanced
        addNext(new AnyOrder(pss.set(i,ps.next()),ps.next().env());
    } }

    if (!advanced()) { //no partial sequence matched nor advanced
      addNext(this,env);
    }

    return false;
} };
```

Figure 6: The AnyOrder class.

where ips is the initial partial sequence of the sequence and as is an ActivationStrategy object that represents the activation strategy of partial sequences. Every ActivationStrategy object has the activate method that is parameterized by a ps partial sequence. This method returns true if ps must active, otherwise the method returns false. We provide three default strategies:

Object	Strategy	It permits to ...
FIRSTANDLAST	First and last partial sequences are only active.	begin more than one sequence at the same time.
LAST	Last partial sequence is only active.	begin only one sequence at the same time.
ALL	all partial sequences are active.	multiplex sequences at any time.

4.2.2 Multiplexer

The multiplexer controls the multiplication of the sequences. The class diagram of Figure 7 shows the Multiplexer abstract class. This class is used to implement sub-classes that provide specific strategies of multiplication of sequences. In this paper, we provide two sub-classes: Multiple and Tracematch. The objects of Multiple class always multiplexes a sequence when it finds an equivalent partial sequence. A case more refined is the Tracematch class because it emulates the behavior found in tracematches [1].

The multiplex method takes a s sequence and a pss array of equivalent partial sequences found in s. The goal of this method is to decide which partial sequences of pss become sequences, which are returned in an ArrayList object. As examples, we present the implementation of the Multiple and Tracematch classes.

Figure 8 shows the Multiple class. The multiplex method creates new sequences of all equivalent partial sequences of pss with ALL object as activation strategy.

The semantic of multiplication of tracematches always permits to

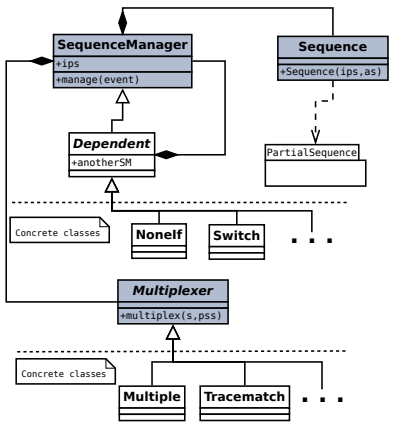


Figure 7: The class diagram of a sequence manager.

```
class Multiple extends Multiplexer {

  ArrayList multiplex(Sequence s,PartialSequence[] pss) {
    ArrayList ss = new ArrayList();
    for (int i = 0; i < pss.length; ++i)
      ss.add(new Sequence(pss[i],ActivationStrategies.ALL));
    return ss;
} }
```

Figure 8: The Multiple class.

```
class Tracematch extends Multiplexer {
  boolean compareEnvs(Env env1,Env env2) {...}
  PartialSequence[] getEquivalents(Sequence s,PartialSequence pt) {...}
  boolean previousIsFirstOfSeq(PartialSequence ps) {...}

  ArrayList multiplex(Sequence s, PartialSequence[] pss) {
    ArrayList ss = new ArrayList();
    for (int i = 0; i < pss.length; ++i) {
      PartialSequence[] epss = getEquivalents(t,pss[i]);
      for (int j = 0; j < epss.length; ++j)
        if (!compareEnvs(epss[j].env(),pss[i].env()) ||
          previousIsFirstOfSeq(pss[i]))
          ss.add(new Sequence(pss[i],ActivationStrategies.ALL));
    }
    return ss;
} };
```

Figure 9: The Tracematch class.

```
class NoneIf extends Dependent {

  ArrayList manage(Event event) {
    ArrayList envs = new ArrayList();
    ArrayList ss = getSequences();
    if (!anotherSM.match(event))
      return super.manage(event);

    ss.removeAll();
    ss.add(new Sequence(ipt,...));
    return new ArrayList();
} };
```

Figure 10: The NoneIf class.

begin a new sequence, and this semantics multiplexes a sequence if two equivalent partial sequences differ in the values of their environments. Figure 9 shows the TraceMatch class. The compareEnvs method verifies if two environments have the same set of values. The getEquivalents method gets all equivalent partial sequences of ps in s. The previousIsFirstOfSeq method verifies if pss[i] is linked to the first partial sequence of the sequence. The multiplex method represents the same semantic of tracematches. The method gets all equivalents partial sequences of every element of pss, and for every equivalent pair, multiplex creates a new sequence if both equivalent partial sequences have different set of values in their environments. A new sequence is also created if pss[i] represents the initial partial sequence of the sequence.

4.2.3 Sequence Manager

The SequenceManager class defines how to manage the matching of all sequences inside a TM. In Figure 7, we can see that SequenceManager has the instance variable ips, which represents the initial partial sequence of the sequence. In addition, this variable is used as a seed to create sequences that begin from the outset. The manage method manages the matching of all sequences. This method takes an event and returns an ArrayList object with the environments of the matched sequences. This class has the Dependent abstract class, which refines the behavior of the manage method for that depending on a specified trace of execution. Besides, this abstract class is used to implement sub-classes that provide specific strategies to manage matching of sequences. For example, the NoneIf class removes all sequences if it matches another trace of execution, or the Switch class that removes all sequences and uses a different initial partial sequence as a seed if it matches another trace of execution. As example, we explain the NoneIf class.

Figure 10 shows the NoneIf class. The manage method, first, verifies whether anotherSM matches or not with the event. In the case that anotherSM matches, the manage method removes all sequences and create a new sequence from ips. If anotherSM does not matches, this method calls the manage method of the super class.

5. CONCLUSION

In this position paper, we described an abstract operational model of TMs and use this abstract model to relate and compare the specific characteristics of existing TMs like PQL, PTQL, Halo, and Alpha. Later, we presented the design of an open TM based on this abstract model. The open model is formulated in a class-based object-oriented setting and follows the design guidelines of open implementations. This model is split into partial sequence and sequence manager model. The first model defines sequences and the second model, taking a partial sequence, defines how to manage and control the matching and multiplication sequences. Both models allows developers to specific strategies of implementation. We showed the openness of our model through concrete and expressive extensions.

Our model has different kinds of challenges. Some challenges are related to its practical adoption. For this reason, as future work, we plan to extend AspectScript [11], an AOP extension of JavaScript, to develop real-world applications that need our model to solve different the issues mentioned in the introduction. Other challenges are related to the understanding of relationships between sequences, multiplexers, and sequence managers. For example, there is a coupling between multiplexers and sequences because a multiplexer needs that sequences uses certain activation strategies to multiplex. Finally, some challenges are related to the static analysis of the sequences due to the highly dynamicity of their operators, *e.g.* consider the random operator that always returns a random partial sequence as next.

6. REFERENCES

[1] Chris Allan, Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Ondrej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. Adding trace matching with free variables to AspectJ. In OOPSLA 2005 [9], pages 345–364. ACM SIGPLAN Notices, 40(11).

[2] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Professional Computing Series. Addison-Wesley, October 1994.

[3] Simon F. Goldsmith, Robert O'Callahan, and Alex Aiken. Relational queries over program traces. In OOPSLA 2005 [9], pages 385–402. ACM SIGPLAN Notices, 40(11).

[4] Stefan Hanenberg, Robert Hirschfeld, and Rainer Unland. Morphing aspects: incompletely woven aspects and continuous weaving. In Karl Lieberherr, editor, *Proceedings of the 3rd ACM International Conference on Aspect-Oriented Software Development (AOSD 2004)*, pages 46–55, Lancaster, UK, March 2004. ACM Press.

[5] Charlotte Herzeel, Kris Gybels, and Pascal Costanza. A temporal logic language for context awareness in pointcuts. In Dave Thomas, editor, *Workshop on Revival of Dynamic Languages*, number 4067 in Lecture Notes in Computer Science, Nantes, France, July 2006. Springer-Verlag.

[6] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William Griswold. An overview of AspectJ. In Jorgen L. Knudsen, editor, *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP 2001)*, number 2072 in Lecture Notes in Computer Science, pages 327–353, Budapest, Hungary, June 2001. Springer-Verlag.

[7] Gregor Kiczales, John Lamping, Cristina V. Lopes, Chris Maeda, Anurag Mendhekar, and Gail Murphy. Open implementation design guidelines. In *Proceedings of the 19th International Conference on Software Engineering (ICSE 97)*, pages 481–490, Boston, Massachusetts, USA, 1997. ACM Press.

[8] Michael Martin, Benjamin Livshits, and Monica S. Lam. Finding application errors and security flaws using PQL: a program query language. In OOPSLA 2005 [9], pages 365–383. ACM SIGPLAN Notices, 40(11).

[9] *Proceedings of the 20th ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2005)*, San Diego, California, USA, October 2005. ACM Press. ACM SIGPLAN Notices, 40(11).

[10] Klaus Ostermann, Mira Mezini, and Christoph Bockisch. Expressive pointcuts for increased modularity. In Andrew P. Black, editor, *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 3586 of *LNCS*, pages 214–240. Springer-Verlag, 2005.

[11] Rodolfo Toledo, Paul Leger, and Éric Tanter. AspectScript: Expressive aspects for the Web. In *Proceedings of the 9th ACM International Conference on Aspect-Oriented Software Development (AOSD 2010)*, Rennes and Saint Malo, France, March 2010. ACM Press. To Appear.

Specifying and Exploiting Advice-Execution Ordering using Dependency State Machines*

Eric Bodden
Software Technology Group
Technische Universität Darmstadt, Germany
bodden@acm.org

ABSTRACT

In this paper we present Dependency State Machines, an annotation language that extends AspectJ with finite-state machines that define the order in which pieces of advice must execute to have a visible effect. Dependency State Machines facilitate the automatic verification and optimization of aspects, but also program understanding.

In this work we present the syntax and semantics of Dependency State Machines and one possible use case of Dependency State Machines: program understanding. We explain how a set of three static program analyses can exploit the information that Dependency State Machines carry to remove advice-dispatch code from program locations at which dispatching the advice would have no effect. Dependency State Machines hereby help to abstract from the concrete implementation of the aspect, making the approach compatible with a wide range of aspect-generating monitoring tools.

Our extensive evaluation using the DaCapo benchmark suite shows that our approach can pinpoint to the user exactly the program locations at which the aspect's execution matters in many cases. This is particularly useful when the aspect's purpose is to identify erroneous execution sequences: in these cases, the program locations that our analysis pinpoints resemble possible points of program failure.

Categories and Subject Descriptors

D.3.4 [**Programming Lang.**]: Processors—*Optimization*

General Terms

Experimentation, Languages, Performance

Keywords

Domain-specific aspect languages, compilation and static program analysis, runtime verification

*The author conducted this work as a PhD student at McGill University, under supervision of Laurie Hendren.

FOAL'10, March 15, 2010, Rennes and Saint Malo, France.

1. INTRODUCTION

Pieces of advice are often inter-dependent in the sense that the execution of one piece of advice will only have an effect before or after the execution of another. This is especially true when the aspect that declares these pieces of advice expresses a state-based runtime monitor. For instance, consider the example aspect in Figure 1, which issues an error message when writing to a disconnected connection. The pieces of advice in this aspect (lines 4–18) monitor `disconnect`, `reconnect` and `write` events on a connection object. The aspect issues an error message when a connection is disconnected and then written to without an intervening reconnect. Figure 2 shows the monitor that this aspect implements in the form of a finite-state machine that issues the error message when reaching its accepting state. It is important to realize that the three pieces of advice in this aspect are inter-dependent: the effect of executing

```
1  aspect ConnectionClosed {
2      Set closed = new WeakIdentityHashSet();
3
4      dependent after disconnect(Connection c) returning:
5          call(* Connection.disconnect()) && target(c) {
6          closed.add(c);
7      }
8
9      dependent after reconnect(Connection c) returning:
10         call(* Connection.reconnect()) && target(c) {
11         closed.remove(c);
12     }
13
14     dependent after write(Connection c) returning:
15         call(* Connection.write(..)) && target(c) {
16         if(closed.contains(c))
17             error("May not write to "+c+", as it is closed!");
18     }
19
20
21     dependency{
22         disconnect, write, reconnect;
23          initial  connected: disconnect -> connected,
24                              write -> connected,
25                              reconnect -> connected,
26                              disconnect -> disconnected;
27                   disconnected: disconnect -> disconnected,
28                              write -> error;
29         final    error: write -> error;
30     }
31 }
```

Figure 1: Monitoring aspect "ConnectionClosed", annotated with Dependency State Machine

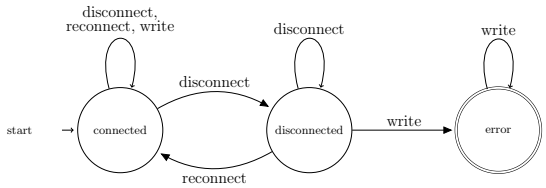


Figure 2: Finite-state machine for "ConnectionClosed" expl.

one piece of advice depends on whether or not other pieces of advice executed already. For instance, the `write` advice will issue an error message for a connection `c` if and only if `disconnect` executed on `c` already, and `reconnect` was not executed in between. Further, line 17 is the only line that has an effect that is visible outside the aspect.

When weaving such aspects into a program with modern aspect compilers like ajc [1] and abc [4], the compilers will report to the user *all* the joinpoint shadows at which the individual pointcuts that the pieces of advice in this aspect refer to could potentially match a joinpoint at runtime. As our experiments show, the sheer number of shadows that such compilers report makes it very hard for programmers to reason about the effect of these aspects. In the ConnectionClosed example, a programmer would have a hard job trying to determine through manual inspection whether or not the program can violate the ConnectionClosed property.

Fortunately, as we show in this work, many of these shadows are "irrelevant" in the sense that, when an irrelevant shadow matches a joinpoint at runtime, then the dispatch of the piece of advice that induced this shadow will never have any effect at this program point. For instance, in the ConnectionClosed example, assume a `write` shadow at a program point at which it is known that the connection that the shadow refers to must be in state "connected". As the state machine in Figure 2 and the aspect code show, the `write` shadow will have no effect in this state. In the state machine, the `write` transition loops, in the aspect code, executing the `write` advice will do nothing because the `if`-check that the body contains must evaluate to `false`.

In this work, we present Dependency State Machines, an annotation language that extends AspectJ and which makes such inter-advice dependencies explicit. In particular, a Dependency State Machine describes the order in which pieces of advice have to execute so that the execution of these pieces of advice, in combination, has an effect that is visible outside the aspect itself. Lines 21–30 in Figure 1 show the appropriate Dependency-State-Machine annotation for the ConnectionClosed example. As the reader can see, we deliberately kept the syntax simple: the annotation directly encodes the appropriate finite-state-machine representation (Figure 2) in a textual format. Line 22 enumerates the alphabet which this state machine is defined over. Every symbol name in this line refers to a named "dependent" piece of advice in the same aspect. Lines 23–29 enumerate all states, along with their outgoing transitions.

If a programmer knows the transition structure of their monitoring aspect, then the programmer can write these annotations by hand. However, many programmers use runtime-monitoring tools to generate such aspects automatically from formal property specifications. In this case, the formal specifications often contain enough information already such that the aspect-generating monitoring tool can automatically generate the appropriate dependency annotations, too. Many monitoring tools use finite-state machines as an internal monitor representation [3, 13, 7, 21], which makes generating the annotations even easier. In the future, researchers could also develop tools that generate Dependency State Machines directly from aspects. However, note that this would involve analyzing Turing-complete aspect code. Therefore, such approaches would always only be able to generate Dependency State Machines for a subset of well-structured aspects.

Once an aspect has been enriched with dependency annotations, tools can exploit the annotations for different purposes. We believe that Dependency State Machines potentially enable a wide range of static analyses and optimizations. In this work, however, we focus on using Dependency State Machines to improve program understanding. We present a flow-sensitive static whole-program analysis, called "Nop-shadows Analysis", that can tell apart irrelevant shadows ("nop shadows") from relevant shadows in many cases. The Nop-shadows Analysis builds on two flow-insensitive analyses that we published previously. These earlier analyses had no information about the order in which pieces of advice must execute, which makes them less precise. In this paper, we show that the Nop-shadows Analysis significantly improves over the results of the earlier analyses and that the combination of all three analyses can significantly reduce the number of shadows that a programmer has to consider when attempting to reason about the aspect's effect.

We validated our approach by applying all three analyses to the 120 combinations of twelve AspectJ aspects (annotated with Dependency State Machines) with ten benchmark programs of the DaCapo Benchmark Suite [6]. As our results show, our analysis successfully identifies a large fraction of shadows as irrelevant. In combination with our two previously published analyses, our novel analysis successfully identified 36020 of 39194 shadows in our benchmark set as irrelevant, i.e., a fraction of 92%. In other words, after applying our analyses, on average, a user would only have to consider about 8% of all shadows to determine where the aspect may have a visible effect. For more than half of the combinations, our analyses were able to show that the aspect has no effect at all on the program's execution. Because our aspects detect erroneous executions, we expect them to have no effect for correct programs. To summarize, this paper presents the following original contributions:

- The syntax and semantics of Dependency State Machines, a novel AspectJ language extension that encodes the order in which pieces of advice must execute to have a visible effect.
- The idea of using Dependency State Machines to improve program understanding by identifying and eliminating irrelevant joinpoint shadows, and a static program analysis that implements these concepts.
- A set of experiments that show that this program analysis can significantly reduce the number of joinpoint shadows that programmers need to consider when trying to reason about their aspects' effects.

Section 2 explains the syntax of Dependency State Machines. In Section 3 we explain our semantics of Dependency State Machines. Section 4 outlines three static analyses that exploit these semantics to identify "irrelevant" joinpoint shadows. We present benchmark results in Section 5, discuss related work in Section 6 and conclude in Section 7.

2. SYNTAX OF DEPENDENCY STATE MACHINES

Figure 1 already demonstrated our language extension using the ConnectionClosed example. Line 22 establishes the alphabet that the state machine is evaluated over. Every symbol in the alphabet refers to a named "dependent" piece of advice in the same aspect. In our language extension, only pieces of advice that are declared as "dependent" can have names. Other pieces of advice have no names and execute with AspectJ's standard semantics. Lines 23–29 enumerate all states in the state machine in question, and for each state enumerate further a (potentially empty) list of outgoing transitions. An entry "`s1: l -> s2`" reads as "there exists an `l`-transition from `s1` to `s2`". In addition, a programmer can mark states as `initial` or `final`, i.e., accepting. We give the complete syntax for Dependency State Machines in Figure 3, as a syntactic extension to AspectJ.

According to the semantics that we will give to Dependency State Machines, the dependency declaration in the ConnectionClosed example states that any piece of `disconnect`, `write` or `reconnect` advice must execute on a connection `c` whenever not executing this piece of advice on `c` would change the set of joinpoints at which the Dependency State Machine reaches its final state on `c`. (More on the semantics later.) Note, however, that the advice references in line 22 omit the variable name `c` of the connection: we just wrote `disconnect, write, reconnect`. We can do so because, by default, a dependency annotation infers variable names from the formal parameters of the advice declarations that it references (lines 4, 9 and 14 in the example). This means that the alphabet declaration in line 22 is actually a short hand for the more verbose form `disconnect(c), write(c), reconnect(c)`.

The semantics of variables in dependency declarations is similar to unification semantics in logic programming languages like Prolog [14]: The same variable at multiple locations in the same dependency refers to the same object. For each advice name, the dependency infers variable names in the order in which the parameters for this advice are given at the site of the advice declaration. Variables for return values from `after returning` and `after throwing` advice are appended to the end. For instance, the following advice declaration would yield the advice reference createIter(c,i).

```
dependent after createIter(Collection c) returning(Iterator i):
  call(* Collection.iterator()) {}
```

We decided to allow for this kind of automatic inference of variable names because both code-generation tools and programmers frequently seem to follow the convention that equally-named advice parameters are meant to refer to the same objects. That way, programmers or code generators can use the simpler short-form as long as they follow this convention. Nevertheless the verbose form can be useful in rare cases. Assume the following piece of advice:

```
dependent before detectLoops(Node n, Node m):
    call(Edge.new(..)) && args(n,m) {
    if(n==m) { System.out.println("No loops allowed!"); }}
```

This advice only has an effect when `n` and `m` both refer to the same object. However, due to the semantics of AspectJ, the advice cannot use the same name for both parameters—the inferred annotation would be detectLoops(n,m). The verbose syntax for dependent advice allows us to state nevertheless that for the advice to have an effect, both parameters actually have to refer to the same object, say `k`: **dependency**{detectLoops(k,k); ... }.

2.1 Type-checking Dependency State Machines

After parsing, we impose the following semantic checks:

- A piece of advice carries a name if and only if it carries also a `dependent` modifier.
- Every advice must be referenced only by a single declaration of a Dependency State Machine.
- The state machine must have at least one initial and at least one final state.
- The listed alphabet may contain every advice name only once, i.e., declares a set.
- The names of states must be unique within the dependency declaration.
- Transitions may only refer to the names of advice that are named in the alphabet of the dependency declaration, and to the names of states that are also declared in the same dependency declaration.
- Every state must be reachable from an initial state.
- If the verbose form for advice references is used:
  - The number of variables for an advice name equals the number of parameters of the unique advice with that name, including the after-returning or after-throwing variable. (inference ensures this)
  - Advice parameters that are assigned equal names have compatible types: For two advice declarations `a(A x)` and `b(B y)`, with `a(p)` and `b(p)` in the same dependency declaration, `A` is cast-convertible [18, §5.5] to `B` and vice versa.
  - Each variable should be mentioned at least twice inside a dependency declaration. If a variable v is only mentioned once we give a warning because in this case the declaration states *no* dependency with respect to v. The warning suggests to use the wildcard "`*`" instead. Semantically, `*` also generates a fresh variable name. However, by stating `*` instead of a variable name, the programmer acknowledges explicitly that the parameter at this position should be ignored when resolving dependencies.

Note that these checks are very minimal and allow for a large variety of state machines to be supplied. For instance, we do allow multiple initial and final states. We also allow the state machine to be non-deterministic. The state machine can have unproductive states from which no final state can be reached, and the state machine even does not have to be connected, i.e. it may consist of multiple components which are not connected by transitions. In this case, the state machine essentially consists of multiple state machines that share a common alphabet. Note that we forbid multiple dependency declarations to reference the same piece of advice: because these dependency declarations could use different alphabets the semantics would be unclear.

Modifier ::= "public" | "synchronized" | ... | **"dependent"**

AdviceDecl ::= *Modifier** [*RetType*] *BefAftAround* ***AdviceName***
"(" [*ParamList*] ")" [*AftRetThrow*] ":" *Pointcut Block*

AdviceName ::= ID

AspectMemberDecl **::=** *AdviceDecl* | ... | *DependencyDecl* | ***DependencySMDecl***

***DependencySMDecl* ::= "dependency" "{" *AdviceRefList* ";" *StateList* ";" "}"**

AdviceRefList* ::= *AdviceRef* | *AdviceRef* "," *AdviceRefList

***AdviceRef* ::= *AdviceName* | *AdviceName* "(" *VarList* ")"**

VarList* ::= *VarName* | *VarName* "," *VarList

***VarName* ::=** ID | **"*"**

StateList* ::= *State* | *State StateList

State* ::= *StateModifier* *Identifier* [":" ***TransitionList***] ";"

***StateModifier* ::= "initial" | "final"**

TransitionList* ::= *Transition* | *Transition* "," *TransitionList

***Transition* ::=** *Identifier* "->" *Identifier*

Figure 3: Syntax of Dependency State Machines, as extension (shown in boldface) to the syntax of AspectJ

3. SEMANTICS OF DEPENDENCY STATE MACHINES

We define the semantics of a Dependency State Machine as an extension to the usual advice-matching semantics of AspectJ [19]. Let $\mathcal{A}$ be the set of all pieces of advice and $\mathcal{J}$ be the set of all joinpoints that occur on a given program run. Consistent with our previous work on Dependent Advice [9], we model advice matching in AspectJ as a function *match* that we regard as given by the underlying AspectJ compiler:

$$match: \quad \mathcal{A} \times \mathcal{J} \quad \rightarrow \quad \{\beta \mid \beta : \mathcal{V} \rightharpoonup \mathcal{O}\} \cup \{\bot\}.$$

For each pair of advice $a \in \mathcal{A}$ and joinpoint $j \in \mathcal{J}$, *match* returns $\bot$ in case a does not execute at j. If a does execute then *match* returns a variable binding β, a mapping from a's parameters to objects ({ } for parameter-less advice).

Based on this definition, we informally demand for any *dependent* piece of advice a, that a only has to execute when it would execute under AspectJ's semantics *and* when *not* executing a at j would change the set of joinpoints at which the Dependency State Machine reaches its final state for a binding "compatible" with β. (We define this term later.)

3.1 Semantics by example

Figure 4 contains a small example program that we use to explain the intuition behind this semantics. The example program violates the ConnectionClosed property in lines 5 and 7 by first disconnecting the connection o(`c1`) and then writing to o(`c1`). (For any variable `v`, we use o(`v`) to refer to the object that `v` references.) The joinpoint shadows [23] at these two lines are also the only two shadows in the program that the ConnectionClosed monitoring aspect from Figure 1 must monitor so that this aspect correctly issues its error message at runtime. In particular, since the monitor starts off in its initial state "connected", the `write` event at line 4 has no impact on the monitor's state: the monitor loops on state "connected", and hence we call the `write` shadow at this line "irrelevant". Similarly, at line 6, the monitor is guaranteed to be in the "closed" state. Monitoring further

```
1  public static void main(String args[]) {
2      Connection  c1 = new Connection(args[0]),
3                  c2 = new Connection(args[1]);
4      c1.write(args[2]);   //write(c1)
5      c1.disconnect();     //disconnect(c1)
6      c1.disconnect();     //disconnect(c1)
7      c1.write(args[2]);   //write(c1)
8      c1.disconnect();     //disconnect(c1)
9      c2.write(args[2]);   //write(c2)
10 }
```

Figure 4: Example program

`disconnect` events does not change the automaton state in this situation either. Hence, the `disconnect` shadows at this line is irrelevant as well. The `disconnect` event at line 8 does cause a state change (from "connected" to "closed"), but this state change does not matter: because no `write` event ever follows on o(`c1`), this state change cannot impact the set of future joinpoints at which the Dependency State Machine reaches its final state (because there are none), and hence cannot impact the set of joinpoints at which the runtime monitor will have a visible effect, i.e., will issue its error message. This is true even though another `write` event follows at line 9. This latter `write` event occurs on `c2` and not on `c1`. Because we know that `c2` cannot possibly reference the same object as `c1`, i.e., o(`c1`) $\neq o$(`c2`), this `write` event is not what we call "compatible" with the `disconnect` event at line 8.

3.2 Formal semantics

In our view of AspectJ, pieces of advice are matched against "parameterized traces", i.e., traces that are parameterized through variable bindings. The semantics of state machines are usually defined using words over a finite alphabet Σ. In particular, state machines as such have no notion of variable bindings. In the following, we will call traces over Σ, which are given as input to a Dependency State Machine "ground traces", as opposed to the parameterized trace that the pro-

gram execution generates. We will define the semantics of Dependency State Machines over ground traces. We obtain these ground traces from the parameterized execution trace by projecting each parameterized event onto a set of ground events. This yields a set of ground traces—one ground trace for every variable binding.

Further, we will define the semantics of Dependency State Machines in terms of "events", not joinpoints. Joinpoints differ from events in that joinpoints describe regions in time while events describe atomic points. A joinpoint has a beginning and an end, and code can execute before or after the joinpoint (i.e., at its beginning or end) or instead of the joinpoint. In particular, joinpoints can be nested. For instance, a field-modification joinpoint can be nested in a method-execution joinpoint. Pieces of advice, even "around advice", execute at atomic events before or after a joinpoint. Because these events are atomic, they cannot be nested. Joinpoints merely induce these events[1].

Event. Let j be an AspectJ joinpoint. Then j induces two events, j_{before} and j_{after} which occur at the beginning respectively end of j. For any set $\mathcal{J}$ of joinpoints we define the set $\mathcal{E}(\mathcal{J})$ of all events of $\mathcal{J}$ as:

$$\mathcal{E}(\mathcal{J}) := \bigcup_{j \in \mathcal{J}} \{j_{\text{before}}, j_{\text{after}}\}.$$

In the following we will often just write $\mathcal{E}$ instead of $\mathcal{E}(\mathcal{J})$, if $\mathcal{J}$ is clear from the context.

For any declaration of a Dependency State Machine, the set of dependent-advice names mentioned in the declaration of the Dependency State Machine induces an alphabet Σ, where every element of Σ is the name of one of these dependent pieces of advice. For instance, the alphabet for the ConnectionClosed dependency state machine from Figure 1 would be $\Sigma = \{\text{disconnect}, \text{write}, \text{reconnect}\}$. Matching these pieces of advice against a runtime event e results in a (possibly empty) set of matches for this event, where each match has a binding attached. We call this set of matches the parameterized event $\hat{e}$.

Parameterized event. Let $e \in \mathcal{E}$ be an event and Σ be the alphabet of advice references in the declaration of a Dependency State Machine. We define the parameterized event $\hat{e}$ to be the following set:

$$\hat{e} := \bigcup_{a \in \Sigma} \{(a, \beta) \mid \beta = \mathit{match}(e, a) \wedge \beta \neq \bot\}.$$

Here, $\mathit{match}(e, a)$ is the "usual" matching function that the original AspectJ semantics provides, overloaded for events.

We call the set of all parameterized events $\hat{\mathcal{E}}$:

$$\hat{\mathcal{E}} := \bigcup_{e \in \mathcal{E}} \{\hat{e}\}$$

It is necessary to consider sets of matches because multiple pieces of advice can match the same event. While this is not usually the case, we decided to cater for the unusual cases, too. As an example, consider the Dependency State Machine in the UnusualMonitor aspect in Figure 5a. The aspect defines a dependency between two pieces of advice `a` and `b`. Note that the pointcut definitions of `a` and `b` overlap, i.e. describe non-disjoint sets of program events. The advice

[1]Our notion of events is essentially the same as the notion of joinpoints in the point-in-time joinpoint model that Masuhara, Endoh and Yonezawa proposed earlier [22].

```
1  aspect UnusualMonitor {
2      dependency{
3          a, b;
4          //transitions omitted from example
5      }
6
7      dependent before a(Object x):
8        call(* *(..)) && target(x) { ... }
9
10     dependent before b(Object x):
11       call(* foo (..)) && target(x) { ... }
12 }
```

(a) UnusualMonitor aspect with overlapping pointcuts

```
1 SomeClass v1 = new SomeClass();
2 SomeClass v2 = new SomeClass();
3 v1.foo (); v1.bar (); v2.foo ();
```

(b) Example program

Figure 5: UnusualMonitor aspect and example program

`b` executes before all non-static calls to methods named `foo`. The advice `a` executes before these events too, because, by its definition, it executes before any non-static method call.

Next, assume that we apply this aspect to the little example program in Figure 5b. We show the program's execution trace in the first row of Figure 6 (to be read from left to right). This execution trace naturally induces the parameterized event trace that we show in the second row of the figure: this trace is obtained by matching at any event every piece of advice against this event.

Next we explain how we use projection to obtain "ground traces", i.e. Σ-words, from this parameterized event trace.

Projected event. For every $\hat{e} \in \hat{\mathcal{E}}$ and binding β we define a projection of $\hat{e}$ with respect to β:

$$\hat{e} \downarrow \beta := \{a \in \Sigma \mid \exists (a, \beta_a) \in \hat{e} \text{ such that } \mathit{compatible}(\beta_a, \beta)\}$$

Here, *compatible* is is a relation over bindings as follows:

$$\mathit{compatible}(\beta_1, \beta_2) := \\ \forall v \in (\mathit{dom}(\beta_1) \cap \mathit{dom}(\beta_2)) \;.\; \beta_1(v) = \beta_2(v)$$

In this equation, $\mathit{dom}(\beta_i)$ denotes the domain of β_i, i.e., the set of all variables that β_i assigns a value. This means that β_1 and β_2 are compatible as long as they do not assign different objects to the same variable.

Parameterized and projected event trace. Any finite program run induces a parameterized event trace $\hat{t} = \hat{e}_1 \ldots \hat{e}_n \in \hat{\mathcal{E}}^*$. For any variable binding β we define a set of projected traces $\hat{t} \downarrow \beta \subseteq \Sigma^*$ as follows. $\hat{t} \downarrow \beta$ is the smallest subset of Σ^* for which holds:

$$\forall t = e_1 \ldots e_n \in \Sigma^* : \\ \text{if } \forall i \in \mathbb{N} \text{ with } 1 \leq i \leq n : e_i \in \hat{e}_i \downarrow \beta \text{ then } t \in \hat{t} \downarrow \beta$$

We call traces like t, which are elements of Σ^*, "ground" traces, as opposed to parameterized traces, which are elements of $\hat{\mathcal{E}}^*$.

For our example, the third and fourth row of Figure 6 show the four ground traces that result when projecting this parameterized event trace onto the variable bindings $x = o(\texttt{v1})$ and $x = o(\texttt{v2})$. For $x = o(\texttt{v1})$ we obtain the two traces "aa" and "ba", for $x = o(\texttt{v2})$ we obtain the two traces "a" and "b".

A Dependency State Machine will reach its final state (and the related aspect will have an observable effect, e.g., will is-

execution trace	`v1.foo();`	`v1.bar();`	`v2.foo();`
parameterized trace $\hat{t}$	$\{(a, x = o(\texttt{v1})),$ $(b, x = o(\texttt{v1}))\}$	$\{(a, x = o(\texttt{v1}))\}$	$\{(a, x = o(\texttt{v2})),$ $(b, x = o(\texttt{v2}))\}$
projected ground traces for $\hat{t} \downarrow x = o(\texttt{v1})$	a b	a a	
projected ground traces for $\hat{t} \downarrow x = o(\texttt{v2})$			a b

Figure 6: Traces resulting from code in Figure 5; note that $o(\texttt{v1}) \neq o(\texttt{v2})$

sue an error message) whenever a prefix of one of the ground traces of any variable binding is in the language described by the state machine. This yields the following definition.

Set of non-empty ground traces of a run. Let $\hat{t} \in \hat{\mathcal{E}}^*$ be the parameterized event trace of a program run. Then we define the set *groundTraces*$(\hat{t})$ of non-empty ground traces of $\hat{t}$ as:

$$\mathit{groundTraces}(\hat{t}) := \left(\bigcup_{\beta \in \mathcal{B}} \hat{t} \downarrow \beta \right) \cap \Sigma^+$$

We intersect with Σ^+ to exclude the empty trace. This is because the empty trace cannot possibly cause the monitoring aspect to have an observable effect.

The semantics of a Dependency State Machine

We define the semantics of Dependency State Machines as a specialization of the predicate $\mathit{match}(a, e)$, which models the decision of whether or not the dependent advice $a \in \mathcal{A}$ matches at event $e \in \mathcal{E}$, and if so, under which variable binding. As noted earlier, this predicate *match* is given through the semantics of plain AspectJ. We call our specialization *stateMatch* and define it as follows:

$$\mathit{stateMatch} : \quad \mathcal{A} \times \hat{\mathcal{E}}^* \times \mathbb{N} \quad \rightarrow \quad \{\beta \mid \beta : \mathcal{V} \rightharpoonup \mathcal{O}\} \cup \{\bot\}$$

$$\mathit{stateMatch}(a, \hat{t}, i) = \text{let } \beta = \mathit{match}(a, e) \text{ in } \begin{cases} \beta & \text{if } \beta \neq \bot \wedge \exists t \in \mathit{groundTraces}(\hat{t}) \\ & \quad \text{such that } \mathit{necessaryShadow}(a, t, i) \\ \bot & \text{else} \end{cases}$$

As we can see, *stateMatch* takes as arguments not only the piece of advice for which we want to determine whether it should execute at the current event, but also the entire parameterized event trace $\hat{t}$, and the current position i in that event trace. Note that $\hat{t}$ contains also future events that are yet to come. This makes the function *stateMatch* undecidable. This is intentional. Even though there can be no algorithm that decides *stateMatch* precisely, we can derive static analyses that approximate all possible future traces. The function *necessaryShadow* mentioned above is a parameter to the semantics that can be freely chosen, as long as it adheres to a certain soundness condition that we define next. We say that a static optimization for Dependency State Machines is sound if it adheres to this condition.

Soundness condition.

The soundness condition will demand that an event needs to be monitored if we would miss a match or obtain a spurious match by not monitoring the event. A Dependency State Machine $\mathcal{M}$ matches, i.e., causes an externally observable effect after every prefix of the complete execution trace that is in $\mathcal{L}(\mathcal{M})$, the language that $\mathcal{M}$ accepts.

Set of prefixes. Let $w \in \Sigma^*$ be a Σ word. We define the set *pref*(w) as:

$$\mathit{pref}(w) := \{p \in \Sigma^* \mid \exists s \in \Sigma^* \text{ such that } w = ps\}$$

Matching prefixes of a word. Let $w \in \Sigma^*$ be a Σ word and $\mathcal{L} \subseteq \Sigma$ a Σ language. Then we define the matching prefixes of w (with respect to $\mathcal{L}$) to be the set of prefixes of w in $\mathcal{L}$:

$$\mathit{matches}_{\mathcal{L}}(w) := \mathit{pref}(w) \cap \mathcal{L}$$

We will often write $\mathit{matches}(w)$ instead of $\mathit{matches}_{\mathcal{L}}(w)$ if $\mathcal{L}$ is clear from the context.

As before, the predicate *necessaryShadow* can be freely chosen, as long as it adheres to the following soundness condition:

Soundness condition. Let $\mathcal{L} := \mathcal{L}(\mathcal{M})$. For any sound implementation of *necessaryShadow* we demand:

$$\begin{aligned} &\forall a \in \Sigma \quad \forall t = t_1 \dots t_i \dots t_n \in \Sigma^+ \quad \forall i \in \mathbb{N} : \\ &\quad a = t_i \wedge \\ &\qquad \mathit{matches}_{\mathcal{L}}(t_1 \dots t_n) \neq \mathit{matches}_{\mathcal{L}}(t_1 \dots t_{i-1} t_{i+1} \dots t_n) \\ &\qquad\qquad \longrightarrow \mathit{necessaryShadow}(a, t, i) \end{aligned}$$

The soundness condition hence states that, if we are about to read a symbol a, then we can skip a if the monitoring aspect would have an observable effect when processing the complete trace t just as often (and at the same points in time) as it would when processing the partial trace where $t_i = a$ is omitted.

4. IDENTIFYING RELEVANT JOINPOINT SHADOWS

In this section, we outline how we use Dependency State Machines to identify "relevant joinpoint shadows", i.e., shadows that may cause the aspect to have a visible effect at runtime. We first explain how our novel flow-sensitive analysis, the Nop-shadows Analysis, identifies "nop shadows": shadows that have no such effect. The "relevant" shadows are then all shadows that the analysis does not classify as "nop shadows". In Section 4.2 we then describe how the Nop-shadows Analysis improves over two analyses that we published previously, and we explain the added benefit of Dependency State Machines over our earlier approach. Section 4.3 gives the most important implementation details.

4.1 Nop-shadows Analysis

We based the Nop-shadows Analysis entirely on our semantics of Dependency State Machines. This semantics states that a dependent advice must be dispatched on some variable binding β if not dispatching the advice would alter the set of events (or joinpoints) at which the monitor reaches its final state for a binding that is compatible with β. The Nop-shadows Analysis exploits this definition by computing

an equivalence relation between states of the Dependency State Machines. This relation allows the analysis to identify "nop shadows" as shadows that only switch between equivalent states. We say that two states q_1 and q_2 are equivalent at a joinpoint shadow s, and write $q_1 \equiv_s q_2$ if, given all possible execution paths that may lead up to s and all possible continuations of the execution after s, the fact whether the monitor is in state q_1 or in state q_2 at s does *not* impact when the Dependency State Machines reaches its final state on these possible continuations. The analysis uses points-to and alias information to disambiguate states for different variable bindings.

Given this equivalence relation, we can then identify shadows s that only switch between "equivalent" states on all possible executions that lead through s. By definition of our semantics of Dependency State Machines we know that dispatching a piece of advice a at such a shadow s would have no effect. We exploit this fact in two different ways. Firstly, we filter the shadow from the list of shadows that is displayed to the user after weaving. This aids the programmer in reasoning about the effects that the aspect may have. Secondly, we remove all advice-dispatch code from this shadow, potentially speeding up the execution of the woven program.

Consider again the example that we gave in Figure 4. We first focus on the `write` shadow at line 4. Given the only possible execution path that leads up to this line, we know that the Dependency State Machine must be in state "connected" when reaching the line. We also know that a `write` transition leads from "connected" back to "connected" only, i.e., the transition loops. State "connected" is obviously equivalent to itself: $q_1 = q_2$ implies $q_1 \equiv_s q_2$. Therefore, the Nop-shadows Analysis can safely disable the advice dispatch at the shadow at line 4. When identifying such a "nop shadow" and disabling the advice dispatch at this shadow, we re-iterate the Nop-shadows Analysis, this time under the new assumption that no advice will be dispatched at the shadow. During this re-iteration, the analysis will disable the `write` shadow at line 9, and either of the `disconnect` shadows at line 5 or 6, depending on which one is analyzed first, and the `disconnect` shadow at line 8. This last shadow at line 8 is interesting in the sense that it switches between equivalent states that are not equal, i.e., we have $q_1 \equiv_s q_2$ although $q_1 \neq q_2$. At this shadow, the non-deterministic Dependency State Machine is simultaneously in states "connected" and "error". From these states, the `disconnect` transition moves into state "disconnected". Although this is definitely not the same internal state, the state "disconnected" is equivalent to both other states *given all possible continuations*, i.e., given all executions that could follow line 8.

Computing the appropriate equivalence relation requires both a forward and a backward-analysis component: the forward component computes equivalencies between states "with respect to the past", while the backward analysis computes equivalencies "with respect to the future", i.e., with respect to the possible continuations. The forward-analysis component works by propagating through the program the states of a determinized version of the original Dependency State Machine $\mathcal{M}$. The backward-analysis component is an exact dual of the forward one: it propagates backwards through the program the states of a determinized version of the inverted state machine of $\mathcal{M}$. To obtain an efficient implementation, our analysis uses flow-sensitive information on an intra-procedural, i.e., per-method level only, and uses a coarse grain flow-insensitive abstraction at method boundaries. Space limitations prevents us from explaining the Nop-shadows Analysis any further. The author's dissertation [8, Section 5.2] explains the analysis in all detail.

4.2 Dependent Advice and previously published analysis stages

In previous work [9], we proposed "Dependent Advice", an AspectJ language extension that, similar to Dependency State Machines, expresses inter-dependencies between pieces of advice. Although both approaches share some ideas, Dependency State Machines improve over Dependent Advice in several ways. The most important improvement is that Dependency State Machines, unlike Dependent Advice, encode the order in which pieces of advice are meant to execute. The Nop-shadows Analysis from above makes heavy use of this information by propagating the state of Dependency State Machines through the program according to their transition tables, which expresses the execution order.

Our earlier approach, Dependent Advice, encoded no such information: a correct Dependent-Advice declaration for our ConnectionClosed example property would be the following.

dependency{ **strong** disconnect, write; **weak** reconnect; }

This declaration follows the syntax that we proposed in earlier work. The declaration states that `disconnect` and `write` share a "strong" dependency. This means that `disconnect` only needs to execute (on a connection `c`) if there is a chance of `write` executing (on `c`) as well, *and* the other way around. The additional "weak" reference to reconnect states that, if the strong dependency is fulfilled, i.e., if both `disconnect` and `write` may execute on the same connection `c` then `reconnect` has to be enabled on `c` as well, but *not* the other way around.

In our earlier work, we presented two flow-insensitive static program analyses that make use of this information. The first analysis, the Quick Check, uses syntactic information only, that we can obtain directly through the weaving process. In our example, if the analysis finds that a program disconnects and reconnects connections but never writes to any connection, i.e., there is no `write` shadow, then this program cannot fulfil the dependency, and hence the entire aspect can have no visible effect for this program.

The second analysis stage, the Orphan-shadows Analysis, performs the same check, but on a per-object basis. This stage uses a flow-insensitive, context-sensitive points-to analysis [26] to disambiguate pointer references. This allows the analysis to decide which joinpoint shadows could potentially refer to the same objects. The analysis then uses this information as follows. In our example, if the program disconnects a particular connection `c` but never writes to `c`, then for this `c` the dependency is not fulfilled and therefore one does not need to monitor any `disconnect`, `reconnect` or `write` events on this connection.

We specifically designed Dependency State Machines in such a way that they are backward compatible to Dependent Advice in the following way. In our previous work we described an algorithm "*genDeps*", which generates Dependent-Advice declarations from any given finite-state machine. We took care to define the semantics of Dependency State Machines in such a way that one can apply *genDeps* directly to any Dependency State Machine to obtain a set of Dependent-

Advice declarations. The flow-insensitive analyses that we proposed earlier can then directly operate on these declarations. This means, that for our ConnectionClosed example, one could obtain the Dependent-Advice declaration that we mentioned above simply by applying the *genDeps* algorithm to the Dependency State Machine from Figure 1.

In our view, Dependency State Machines are easier to understand than Dependent Advice because their semantics follow the semantics of finite-state machines, which are well understood. Especially, it is safe to assume that most programmers are familiar with the basic semantics of finite-state machines. For Dependent Advice, the semantics are less obvious. Therefore, Dependency State Machines combine two advantages: they encode richer information, and nevertheless they are potentially easier to use.

4.3 Implementation

The flow-insensitive Quick Check and the Orphan-shadows Analysis generally finish faster than the more involved flow-sensitive Nop-shadows Analysis. Therefore, it is a good idea to apply the Nop-shadows Analysis only after the Quick Check and the Orphan-shadows Analysis have been applied first.

We implemented Dependency State Machines as an extension to the AspectBench Compiler [4] (abc) that builds on exactly these ideas. Our abc extension first extracts Dependency State Machines from the aspect definitions. Then it uses the *genDeps* algorithm to generate Dependent-Advice declarations from these Dependency State Machines. Next, we instruct abc to weave all aspects (whether they contain dependency declarations or not) into the given program. Our extension then applies both the Quick Check and the Orphan-shadows Analysis from previous work. These analyses only access the generated flow-insensitive Dependent-Advice declarations, no Dependency State Machines. If potentially relevant shadows remain after applying these two stages, then our extension invokes the Nop-shadows Analysis. This analysis is flow-sensitive, and therefore it accesses the Dependency State Machines to extract the information about the advice-execution ordering that the state machine's transition structure expresses. Every analysis stage may identify "irrelevant" shadows. In the end, our extension instructs abc to un-weave and re-weave the program, this time with all "irrelevant shadows" disabled.

Our abc extension contains two front-ends that both create an internal representation of the Dependency State Machines that the given program contains. One front end is implemented as an extension to the abc-internal parser. We use this front end to parse AspectJ source files that contain declarations of Dependency State Machines. The second front end that we provide creates the internal representation of the Dependency State Machines directly from a given set of tracematches [3]. Tracematches is another AspectJ language extension that allows programmers to define an AspectJ-based runtime monitor in a declarative way, using a regular-expression syntax. The aspects that abc generates from tracematches are never written to disc. abc instead generates these aspects in the form of Jimple [28] three-address code, an internal representation of the compiler, and then weaves the aspects into the designated program directly on this representation. Our tracematch front end therefore extracts the state machine directly from abc's internal representation of the tracematch.

5. EXPERIMENTS

To validate our approach, we defined a set of twelve monitoring aspects as tracematches. All aspects monitor for violations of safety properties. Table 1 explains the properties that these aspects monitor. We then applied the twelve aspects to ten benchmark programs of the DaCapo benchmark suite [6]. This lead to 120 aspect/benchmark combinations.

We were interested in answering two research questions. The first question evaluates how much our approach improves over previous work: (1) How effective is the Nop-shadows Analysis in identifying irrelevant shadows when compared to our two previously published static analyses that consider flow-insensitive dependency information from Dependent Advice only? The second question evaluates our approach from the user's point-of-view: (2) How effective is the overall approach, i.e., the combination of all three analysis stages (Quick Check, Orphan-shadows Analysis and our novel Nop-shadows Analysis) in telling apart irrelevant shadows from potentially relevant shadows.

To answer both questions, we decided to compare the number of shadows that our approach fails to identify as irrelevant, i.e., the number of potentially relevant shadows, to two different baselines: (1) the number of shadows that remain potentially relevant after applying the first two analysis stages, and (2) the total number of shadows that a compiler that is unaware of our dependency annotation and conducts no static analysis would present to the user. Table 2 summarizes our analysis results with respect to both baselines.

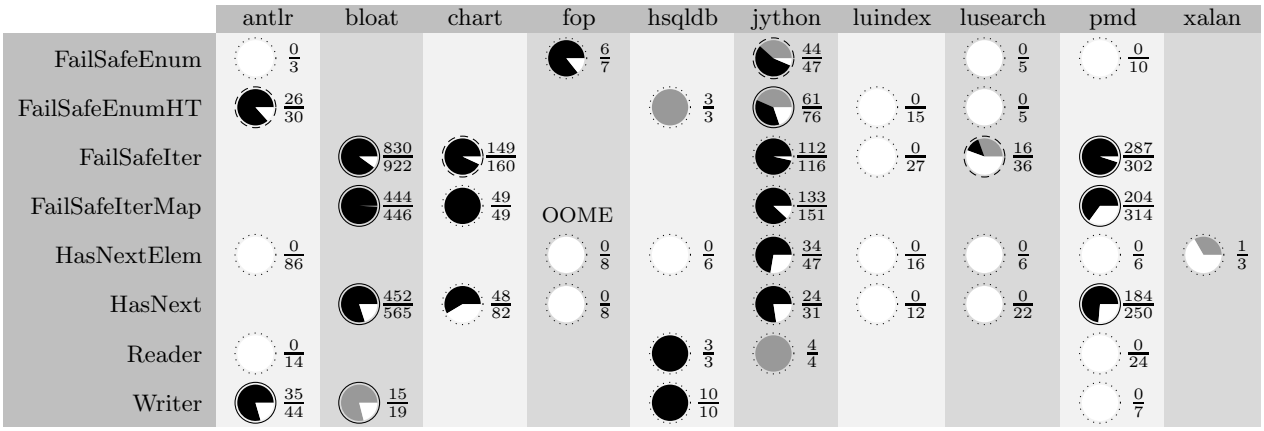
5.1 Analysis precision compared to previously published analyses

Table 2a shows the fraction of shadows that the Nop-shadows Analysis identified as irrelevant, where the baseline of this fraction is the number of potentially relevant shadows after applying the Quick Check and the Orphan-shadows Analysis. From this table we omitted those entries (and, where applicable, entire lines) for which the number of potentially relevant shadows was zero already after just applying these two analysis stages. The fraction of shadows that the Nop-shadows Analysis identified as nop shadows appears in white. For some combinations where only few shadows remained enabled after applying the analysis, we inspected these shadows manually. In gray we show the fraction of shadows that we manually determined to be relevant. The remaining black slices represent shadows that remain active even after analysis, either due to analysis imprecision or because they are actually relevant although they were not manually confirmed to be relevant. For the combination fop-FailSafeIterMap (1374 shadows to analyze), our Nop-shadows Analysis ran out of memory although we had provided abc with three gigabytes of heap space. As a research prototype, our analysis is currently not optimized towards low memory consumption.

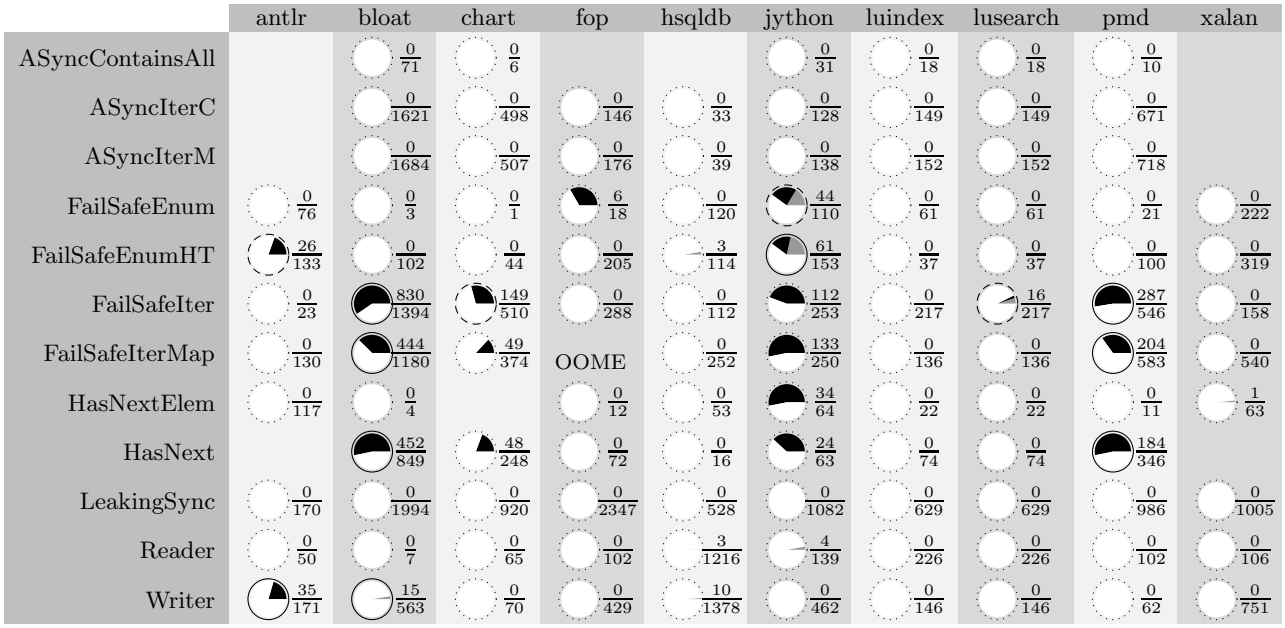
For 18 out of these 43 combinations (41%), our novel Nop-shadows Analysis was able to identify all shadows as irrelevant. Because our aspects monitor safety properties, this means that, in these 18 cases, the analysis proved that the given program cannot possibly violate the given property. These cases appear as all-white circles. In four other cases, shadows remained enabled, but only because they do trigger a property violation. These cases appear as circles that only contain gray or white but no black slices. In other words, the analysis gave exactly the correct result, with no false posi-

property name	description
ASyncContainsAll	synchronize on `d` when calling `c.containsAll(d))` for synchronized collections `c` and `d`
ASyncIterC	only iterate a synchronized collection `c` when owning a lock on `c`
ASyncIterM	only iterate a synchronized map `m` when owning a lock on `m`
FailSafeEnum	do not update a vector while iterating over it
FailSafeEnumHT	do not update a hash table while iterating over its elements or keys
FailSafeIter	do not update a collection while iterating over it
FailSafeIterMap	do not update a map while iterating over its keys or values
HasNextElem	always call hasMoreElements before calling nextElement on an Enumeration
HasNext	always call hasNext before calling next on an Iterator
LeakingSync	only access a synchronized collection using its synchronized wrapper
Reader	do not use a Reader after its InputStream was closed
Writer	do not use a Writer after its OutputStream was closed

Table 1: Relevant typestate properties and their names



(a) Potentially relevant shadows as fraction of shadows that remain after first two analysis stages



(b) Potentially relevant shadows as fraction of total shadows after weaving

Table 2: Irrelevant vs. potentially relevant shadows. White slices represent shadows that the Nop-shadows Analysis identified as irrelevant. Black slices represent shadows that we fail to identify as irrelevant, due to analysis imprecision or because the shadows are relevant. Gray slices represent shadows that we confirmed to be relevant, through manual inspection. The outer rings represent the aspect's runtime overhead after optimizing the advice dispatch. Solid: overhead $\geq 15\%$, dashed: overhead $< 15\%$, dotted: no overhead. OOME = OutOfMemoryException during static analysis

tives, in half of the cases. In three cases, the analysis failed to identify any nop shadow (black circles). In the remaining 18 cases, the analysis identified a sometimes significant amount of irrelevant shadows, but not all.

Our analysis works well on the antlr, fop, hsqldb, luindex, lusearch and xalan benchmarks. Most of the potential false positives (black in the figure) appear only because the benchmarks use reflection. Due to a known deficiency [2], Java's `Cloneable` interface contains no public declaration of a `clone()` method. Therefore, Java's type system may prevent clients from calling `clone()` even on `Cloneable` objects. chart uses reflection to call the `clone()` method on objects that implement the `Cloneable` interface. Because chart clones collections, our points-to analysis has to safely assume that the collections could be of any type, including `EmptySet`, which, as a singleton object, is stored in a static field, causing our analysis to lose all context information. bloat, jython and pmd cause similar problems.

There appear to be only few cases where our analysis is too imprecise because of its design. For example, two actually irrelevant final shadows remain enabled in hsqldb with Reader and Writer. These false positives occur because xalan uses different methods to open, close and write to streams. Our current implementation of the Nop-shadows Analysis uses flow-sensitive information on an intra-procedural level only, and therefore cannot possibly produce precise analysis results in this context. In the future, we plan to extend the Nop-shadows Analysis into a fully inter-procedural version that will treat these cases more precisely.

DaCapo's benchmarks load classes using reflection. Static analyses like ours have to be aware of these classes so that they can construct a sound call graph. We wrote an AspectJ aspect that would print at every call to `forName` and a few other reflective calls the name of the class that this call loads and the location from which it is loaded. We further double-checked with Ondřej Lhoták, who compiled such lists of dynamic classes earlier. We then provided the abc-internal call-graph analysis with this information. The resulting call graph is sound for the program runs that DaCapo performs. A limitation of our approach is that obtaining a call graph that is sound for all runs may be challenging for programs that use reflection.

For eclipse we were unable to determine where dynamic classes are loaded from. eclipse loads classes not from JAR files but from "resource URLs", which eclipse resolves internally, usually to JAR files within other JAR files. abc currently cannot load classes from such URLs and that is why we omit eclipse in our experiments. The jython benchmark generates code at runtime, which it then loads. We did not analyze this code and so made the unsound assumption that this code would not invoke any dependent advice.

5.2 Fraction of potentially relevant shadows over number of all shadows after weaving

Table 2b shows the fraction of shadows that the Nop-shadows Analysis identified as irrelevant, where the baseline of this fraction is the number of all shadows that a compiler without any of our static analysis would usually report to the user. This fraction shows to what extend users can benefit through the use of Dependency State Machines in general, when applying all three of our static analyses in combination, compared to not using Dependency State Machines at all. Opposed to Table 2a, this table shows the important piece of information that, in many cases, the Quick Check and the Orphan-shadows Analysis manage to identify many irrelevant shadows already. Often, these analyses are even sufficient to identify that all shadows are irrelevant, i.e., that the aspect will never have a visible effect. The high amount of white and gray in this table shows that our overall static-analysis approach is very effective in pinpointing to the user the relevant shadows that will cause the aspect to have a visual effect at runtime.

5.3 Reduction of runtime overhead

An added benefit of our analysis is that we can use the analysis result to optimize the advice dispatch, which may reduce the aspect's runtime overhead. Table 2 gives qualitative information about the optimized aspect's runtime overhead through the ring that surround each circle. (The author's dissertation [8] gives the full data.) Interestingly, the number of remaining shadows does not necessarily correspond directly to the resulting runtime overhead. For instance, only 15 out of 563 shadow remain in bloat-Writer, but these shadow executes so often that they cause a runtime overhead of more than 15%. chart-FailSafeIterMap, on the other hand, contains 49 residual shadows, but there is no observable overhead. Altogether, after applying the Nop-shadows Analysis, only nine combinations remain that have a significantly perceivable overhead of more than 15%. Most combinations show zero overhead, five combinations show an overhead of below 15%, which seems negligible in many cases.

6. RELATED WORK

We compare our work to the most related static program analyses and to aspect-oriented model-checking approaches. In addition, we discuss tools that generate AspectJ aspects from high-level specifications and whether these tools could generate Dependency State Machines as well.

6.1 Static program analysis

Whole-program analysis of tracematches. Tracematches [3] is an AspectJ language extension that allows programmers to express finite-state properties using a high-level language that is based on regular expressions. Like Dependency State Machines, tracematches are implemented on top of the AspectBench Compiler. During compilation, the compiler internally reduces tracematches to "normal" AspectJ aspects. Several people [10, 11, 24], including ourselves, have proposed static analyses that exploit the information that tracematches contain to optimize advice dispatch. The Quick Check and the Orphan-shadows Analysis, that we discussed in Section 4, are generalized versions of two similar analyses that we previously implemented specifically for tracematches [10]. The Nop-shadows Analysis that we presented in this paper is defined directly in terms of Dependency State Machines, and we never implemented a tracematch-specific version for it (although we could have). The advantage of Dependency State Machines is not that they improve the analyzability of tracematches or similar formalisms, but rather that Dependency State Machines make existing analyses applicable to aspects in general, no matter whether the aspects were generated from tracematches, from any other high-level specification or even written by hand.

Monitor optimizations. Avgustinov et al. [5] proposed optimizations to the monitoring aspect itself: *Leak elimination* discards monitoring state for objects that have been garbage collected. *Indexing* provides for fast access to partial matches. These optimizations are crucial to make runtime monitoring feasible at all and therefore we enabled them in our experiments when generating aspects from tracematches. JavaMOP [13] and PTQL [17] implement weaker variants of these optimizations.

One big advantage of Dependency State Machines is that they allow researchers to de-couple the optimizations of runtime monitors, i.e., the code that "goes into the advice bodies", from analyzing and optimizing the advice dispatch, based on the order in which these pieces of advice should execute. In theory it would be possible to determine an aspect's transition structure directly through static analysis, without requiring an explicit dependency annotation. However, general AspectJ code is Turing complete, which makes this analysis problem generally undecidable. In particular, the optimizations that aspect-generating monitoring tools conduct can lead to arbitrarily complex aspect code, much more complex than the code that we showed in Figure 1. This makes it very hard for static analyses to re-discover the transition structure directly from the code. Dependency State Machines elegantly solve this problem by simply specifying the transition structure directly in a machine readable format.

6.2 Model Checking

Goldman and Katz [16] propose a model-checking approach that can "once and for all" verify an aspect "relative to its specification", i.e., independently of any specific program that this aspect may be woven into. The authors developed the MAVEN tool that implements this modular aspect-verification mechanism. Like ourselves, the authors assume that the aspect's internal structure can be represented as a finite-state machine. However, unlike us, Goldman and Katz do not state how programmers would communicate this transition structure to the model checker. Our proposed syntax, Dependency State Machines, closed this gap. Another restriction of Goldman and Katz's approach is that the authors assume that one can represent the non-aspect parts of the program as a finite-state machine as well. This is necessary, because the authors model the weaving process through a series on transformations on state machines. In our approach, we make no such assumption. We leave the weaving semantics to standard AspectJ.

It would be an interesting piece of future work to determine whether the semantics that we gave to Dependency State Machines is compatible with the finite-state-machine semantics that Goldman and Katz's approach requires. If so, it should be easily possible to integrate MAVEN with our abc extension.

6.3 Aspect-generating tools

JavaMOP. JavaMOP [13] is an open research framework for generating AspectJ monitoring aspects from several kinds of formal specifications, including Extended Regular Expressions, Past-time and Future-Time Linear Temporal Logic. In previous work [9], Feng Chen has modified the JavaMOP implementation so that it internally generates a finite-state machine from all these formal specification, regardless of the formalism that is used. JavaMOP can directly benefit from Dependency State Machines by annotating the generated aspects with these state machines.

Association aspects and relational aspects. Sakurai et al. [25] proposed *association aspects*, an AspectJ language extension that allows programmers to restrict advice execution to joinpoints involving objects that the programmer explicitly associated with an aspect. A programmer associates an object `o` with an aspect `A` by calling `A.associate(o)`, and releases the association via `A.release(o)`. In earlier work [12] we showed that one can implement *relational aspects*, a variant of *association aspects*, via a syntactic transformation into tracematches. abc implements relational aspects that way, and the implementation automatically benefits from our extension: The optimizations proposed in this paper remove advice dispatch code for an advice contained in an aspect A from locations where the objects involved are known to be either not yet associated with A or to already have been released from A.

S2A, M2Aspects and J-LO. Maoz and Harel proposed S2A, a tool [21] to generate executable AspectJ code from Live Sequence Charts [15] (LSCs). An LSC and its generated aspects can either implement functional aspects of a system, or they can be used for runtime monitoring, reporting error messages when they match. Some of the aspects that S2A generates are history-based, and in fact even implement a finite-state machine. We confirmed with Maoz that S2A could, in principle, generate dependency annotations for these aspects and that they could lead to optimization potential similar to what we observed in our experiments, at least when LSCs are used to specify forbidden scenarios, implemented as runtime monitors. M2Aspects [20] generates AspectJ aspects from scenario-based software specifications, denoted as Message Sequence Charts (MSCs). MSCs are less expressive than LSCs. Hence we believe that one could also modify M2Aspects to generate dependent advice. J-LO, the Java Logical Observer [7, 27] generates AspectJ aspects from formulae written in a special future-time linear temporal logic with free variables. Internally, J-LO represents the formulae using alternating automata. There exists a standard algorithm to convert alternating automata into finite-state machines. J-LO could therefore easily benefit from Dependency State Machines by implementing this conversion and annotating the generated aspect with the appropriate Dependency-State-Machine declaration.

7. CONCLUSIONS AND FUTURE WORK

In this paper we have presented Dependency State Machines, a language extension to AspectJ that expressed the order in which pieces of advice have to occur, so that these pieces of advice, in combination have an effect that is visible outside the declaring aspect. We have shown how to use the information that Dependency State Machines provide to facilitate program understanding. We have outlined a set of static analyses that can identify "irrelevant" joinpoint shadows with high precision. When trying to determine the effects that their aspects may have, programmers do not need to consider such irrelevant shadows.

Nevertheless, we believe that also in fields different from program understanding, Dependency State Machines offer the potential for a lot of exciting research opportunities that researchers could address in the near future. One interesting field of research could be the inference of Dependency State Machines. Our current approach assumes that Dependency

State Machines are present in aspect code, i.e., that either the programmer or some aspect generating tool supplied the state-machine declaration. In many cases, it could be possible to infer these declarations automatically from aspect code or from dynamic executions.

Another interesting research question would be how Dependency State Machines can be used to verify or check an aspect's execution. According to the semantics that we gave in this paper, a Dependency State Machine expresses the order in which pieces of advice must execute to have a visible effect. One could give different semantics to Dependency State Machines, e.g. that a Dependency State Machine describes the order in which pieces of advice are allowed to be called by the surrounding context, i.e., the program which the pieces of advice are woven into. Static or runtime verification could then try to determine, for a particular execution context, whether this context fulfils the aspects execution requirements.

Acknowledgements. This work originated from a collaboration of the author with Feng Chen and his supervisor Grigore Roşu. Tragically, Feng passed away in summer 2009. May he rest in peace and may his legacy not be forgotten. The author conducted this work as a PhD student at McGill University, under supervision of Laurie Hendren. Thanks, Laurie, for all the support you gave me at McGill. This work was supported by NSERC and the Center for Advanced Security Research Darmstadt (www.cased.de).

8. REFERENCES

[1] The AspectJ home page. http://eclipse.org/aspectj/.

[2] Bug-database entry regarding "Cloneable". `http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=4098033`.

[3] C. Allan, P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. Adding Trace Matching with Free Variables to AspectJ. In *International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 345–364. ACM Press, Oct. 2005.

[4] P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, J. Lhoták, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. *abc*: An extensible AspectJ compiler. In *International Conference on Aspect-oriented Software Development (AOSD)*, pages 87–98. ACM Press, Mar. 2005.

[5] P. Avgustinov, J. Tibble, and O. de Moor. Making trace monitoring feasible. In *International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 589–608. ACM Press, Oct. 2007.

[6] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanovic, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA*, pages 169–190. ACM Press, Oct. 2006.

[7] E. Bodden. J-LO - A tool for runtime-checking temporal assertions. Diploma thesis, RWTH Aachen University, November 2005.

[8] E. Bodden. *Verifying finite-state properties of large-scale programs*. PhD thesis, McGill University, June 2009.

[9] E. Bodden, F. Chen, and G. Roşu. Dependent advice: a general approach to optimizing history-based aspects. In *AOSD '09: Proceedings of the 8th ACM international conference on Aspect-oriented software development*, pages 3–14. ACM, 2009.

[10] E. Bodden, L. J. Hendren, and O. Lhoták. A staged static program analysis to improve the performance of runtime monitoring. In *European Conference on Object-Oriented Programming (ECOOP)*, volume 4609 of *Lecture Notes in Computer Science (LNCS)*, pages 525–549. Springer, 2007.

[11] E. Bodden, P. Lam, and L. Hendren. Finding programming errors earlier by evaluating runtime monitors ahead-of-time. In *Symposium on the Foundations of Software Engineering (FSE)*, pages 36–47, New York, NY, USA, 2008. ACM.

[12] E. Bodden, R. Shaikh, and L. Hendren. Relational aspects as tracematches. In *International Conference on Aspect-oriented Software Development (AOSD)*, pages 84–95. ACM Press, Mar. 2008.

[13] F. Chen and G. Roşu. MOP: an efficient and generic runtime verification framework. In *International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 569–588. ACM Press, Oct. 2007.

[14] W. F. Clocksin and C. Mellish. *Programming in Prolog, 5th Edition*. Springer, 2003.

[15] W. Damm and D. Harel. LSCs: Breathing life into message sequence charts. *Formal Methods in System Design*, 19(1):45–80, 1999.

[16] M. Goldman and S. Katz. MAVEN: Modular Aspect Verification. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 308–322, 2007.

[17] S. Goldsmith, R. O'Callahan, and A. Aiken. Relational queries over program traces. In *International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 385–402, Oct. 2005.

[18] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java(TM) Language Specification, Third Edition*. Addison-Wesley Professional, 2005.

[19] E. Hilsdale and J. Hugunin. Advice weaving in AspectJ. In *International Conference on Aspect-oriented Software Development (AOSD)*, pages 26–35. ACM Press, Mar. 2004.

[20] I. H. Krüger, G. Lee, and M. Meisinger. Automating software architecture exploration with M2Aspects. In *SCESM*, pages 51–58. ACM Press, 2006.

[21] S. Maoz and D. Harel. From multi-modal scenarios to code: compiling LSCs into AspectJ. In *Symposium on the Foundations of Software Engineering (FSE)*, pages 219–230. ACM Press, Nov. 2006.

[22] H. Masuhara, Y. Endoh, and A. Yonezawa. A fine-grained join point model for more reusable aspects. *Programming Languages and Systems*, 4279:131–147, 2006.

[23] H. Masuhara, G. Kiczales, and C. Dutchyn. A compilation and optimization model for aspect-oriented programs. In *International Conference on Compiler Construction (CC)*, volume 2622 of *Lecture Notes in Computer Science (LNCS)*, pages 46–60. Springer, Apr. 2003.

[24] N. A. Naeem and O. Lhoták. Typestate-like analysis of multiple interacting objects. In *International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 347–366, New York, NY, USA, 2008. ACM.

[25] K. Sakurai, H. Masuhara, N. Ubayashi, S. Matsuura, and S. Komiya. Association aspects. In *International Conference on Aspect-oriented Software Development (AOSD)*, pages 16–25, Mar. 2004.

[26] M. Sridharan and R. Bodík. Refinement-based context-sensitive points-to analysis for Java. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 387–400, June 2006.

[27] V. Stolz and E. Bodden. Temporal Assertions using AspectJ. In *5th Workshop on Runtime Verification*, volume 144 of *Electronic Notes in Theoretical Computer Science*, pages 109–124, July 2005.

[28] R. Vallée-Rai and L. Hendren. Jimple: Simplifying Java Bytecode for Analyses and Transformations. Technical Report 1998-4, Sable Research Group, July 1998.

Semantic Aspect Interactions and Possibly Shared Join Points

Emilia Katz Shmuel Katz
Computer Science Department
Technion – Israel Institute of Technology
{emika, katz}@cs.technion.ac.il

ABSTRACT

When multiple aspects can share a join-point, they may, but do not have to, semantically interfere. We present an in depth analysis of aspect semantics and mutual influence of aspects at a shared join-point, in order to enable programmers to distinguish between potential and actual interference among aspects at shared join-points. An interactive semi-automatic procedure for specification refinement is described, that will help users define the intended aspect behavior more precisely. Such a refined specification enables modular verification and interference detection among aspects even in the presence of shared join-points.

Keywords

Aspect interference, semantics, specification, shared join-points

1. INTRODUCTION

Multiple aspects, when woven into the same base system, might happen to have common join-points. This possibility gives rise to many important questions and problems, from understanding how the potentially applicable advice pieces should be woven at such a common join-point (as they cannot be applied all at the same time), to conflict detection and resolution, since application of one advice might interfere with the computation or even applicability of another.

In this paper we consider this question in depth, for aspects and systems modeled as state transition diagrams, with specifications given as linear temporal logic (LTL) assumptions and guarantees. As a solution to aid in understanding the implications of shared join-points, we describe an easily automatizable interactive procedure that will help users specify their intentions for aspect behavior in a specific system more precisely, and check whether there is actual interference with respect to this specification. Based on the answers to a series of questions to the user, the LTL specifications of the aspects are automatically augmented. The state transition model is also modified to handle shared join-points. Then the verification and automatic interference checks from [10] can be used to detect subtle cases of interference at shared join-points, or establish that there is no such interference, using the augmented specifications.

Ways to detect shared join-points are described in [14, 15]. Several works study shared join-points as a source of possible conflicts, some (e.g., [14]) even see common join-points as the main source of interference among aspects. A language independent technique [8] makes it possible to check whether an undesired order of aspect application at a shared join-point is possible, where the list of undesired orders has to be explicitly provided by the user. It is implemented in the "Secret" tool for Compose* [2, 13]. However, presenting the undesired orders list requires a thorough analysis of the system by the user, and also might not be able to reflect all the intended behaviors, as at different states different orders of application might be possible. In [1] another tool for checking potential interference at common join-points is described, applicable for the Compose* language. It checks all the possible orders of aspect applications at a common join-point, and declares a conflict if different orders result in different resulting states. This method is fully automatic, but may lead to many false positives (some of which are described later in this paper). An additional tool for aspect interference detection, performing dataflow checks to find out whether one aspect affects variables used in another, is presented in [17]. It can also be used to check interactions at shared join-points, though its scope is broader. Interactions found by this tool are also only potentially harmful.

Weaving techniques for conflict resolution at shared join-points appear in [14, 6, 7]. In [14], a first analysis of types of mutual influence of aspects applied at a shared join-point appears. This analysis is extended in our paper, though used for a different purpose.

The intended semantics of weaving several aspects at a common join-point in a pre-defined order is addressed in papers on the semantics of aspects, such as [16, 4, 5].

However, as described in [12], not all the conflicts at shared join-points can be resolved by a clever weaving. Thus it is important for the user to be able to detect the conflicts and differentiate between real problems and false alarms. Below we show the possible influences of multiple aspects on each other at a join-point, and classify them according to the way they affect the specification of the aspect.

The paper is organized as follows: Our analysis of aspect semantics and types of mutual influence at shared join-points appears in Section 2. In Section 3 we show the questions the answers to which can be automatically processed to add appropriate predicates to a temporal logic specification. Section 4 gives examples of applying the questions. We conclude in Section 5.

2. SEMANTICS OF ASPECT BEHAVIOR AT A COMMON JOIN POINT

In AspectJ, the aspects at a common join-point are applied one after another, and each time before performing an advice the pointcut condition is re-checked. As a result of such a semantics, when a base system arrives at a join-point matched by an aspect A, it is not necessarily the case that the advice of A is immediately executed. It might be the case that other aspects are present in the system that also match this join-point, and it might thus happen that some other advices are executed before the advice of A, changing the state of the system in which A will be applied. Moreover, A's advice might not be executed at all, in case one of the previously executed aspects left the system in a state which is not a join-point of A any more.

Thus the execution of the woven system from the moment it arrives at a join-point matched by some of its aspects is determined not only by the set of matching aspects, but also by the order of their application at this point. So if this order of application is not explicitly prescribed by the user, the non-determinism of aspect application may result in different states.

However, the fact that different orders of advice application lead to different resulting states does not necessarily mean that the aspects semantically interfere. Let us consider the following example, presented in [1]: Several aspects are defined for systems in which messages of type String are sent between objects. Two of these aspects are Logging and Encryption. Both aspects are applied at the same join-points - when a message is sent in the system - and different orders of their application will result in different states of the system. If Logging is executed before Encryption, the logged message will be the original one, otherwise it will be the encrypted message produced by the Encryption aspect. In [1] such a situation is considered interference between the two aspects, but in fact the decision on whether it is interference or not should depend on the aspects' specifications. In our example, the goal of the Encryption aspect is to encrypt every message before it is sent to the server. Consider the following possible specifications of the Logging aspect, described more formally in Section 4:

1. The log should record all the messages as they were originally attempted to be sent by the user, so that the user will be able to view the list of messages (s)he sent.
2. The log should record all the messages as they were actually sent to the server in order to compare the sent messages to the received ones (as received) and verify that no messages got lost or garbled.
3. The goal of the Logging aspect is to measure the network activity of the system. Thus, though the contents of the messages are written to the log, they are of no importance to the user, and what matters is only the number of messages sent and their frequency, e.g. the times of the messages sent and the number of lines in the log.
4. The logging records all the attempts to send a message, even if they are aborted for whatever reason. It logs each message as it was attempted to be sent by the user.

All the cases above can happen in our example system, and different order of application of the two aspects at their common join-point will lead to different resulting states, but not in all the cases above do the aspects interfere. The requirements from the Encrypting aspect are never violated by Logging, no matter in what order they are executed, but in variants (1), (2) and (4) the aspects may interfere: in variants (1) and (4), the goal of Logging will not be reached if the Encrypting aspect is applied first, and in variant (2), applying Encrypting after Logging will cause a problem. However, in variant (3) applying the aspects in any order will not violate the requirements from Logging or from Encrypting, thus there will be no interference. As will be shown later, an Authorization aspect can also be applied, further complicating the situation.

The above example shows the need to analyze possible semantic effects of sharing a join-point more deeply. We consider the AspectJ operational semantics, where first all the places in the code of the base program that are matched by the static part of some aspect's pointcut, are identified. Such places are called *shadow join-points*. Note that a shadow join-point is usually defined by a place in the code of the program, but sometimes can contain additional information. After shadow join-point identification, at each such join-point the weaving order of the potentially applicable aspects is defined (an aspect is considered potentially applicable if the static part of its pointcut matches the current shadow join-point). The weaving order does not have to be defined statically, it can be determined upon arrival of the computation at the join-point. At last, when a computation arrives at a join-point, each of the potentially applicable aspects, one by one and in the previously defined order, is checked for full applicability and immediately executed if indeed applicable (i.e., if both static and dynamic parts of the pointcut are matched by the current state). All the rest of the paper refers to this semantics, and if a different semantics is chosen, different reasoning might be needed. This operational semantics shows the need to reason about the part of computation between the first moment it arrives at some shadow join-point and the moment it leaves this join-point, which includes all the aspect applications performed at the join-point. We need some new terminology to make this reasoning easier. First of all, we need a name for the period of interest:

DEFINITION 1. *A sequence of states $s_1, \ldots, s_k$ in a computation of the woven system is called a* pointcut occurrence *of aspect A if s_1 is the state when a join-point of A is first reached (that is, s_1 is matched by the full pointcut descriptor*

of A, and the previous state is not), and s_k is the state when the computation is about to leave the corresponding shadow join-point, after application of all the appropriate aspect advices according to the current weaving policy (that is, s_k is matched by the static part of the pointcut descriptor of A, and the next state is not).

Two aspects share a join-point if they have at least one overlapping pointcut occurrence. Note that overlapping pointcut occurrences do not have to coincide, as it might be a case that an execution arrives at a state s matched by the pointcut descriptor of aspect B, and by the static part of pointcut descriptor of A and B, but not matched by the dynamic part of A's pointcut descriptor, and only the execution of aspect B at s will result in a state in which both static and dynamic parts of A's pointcut hold. In such a case the pointcut occurrence of A will be contained in the pointcut occurrence of B, but not vice versa. For example, consider the case of aspects A and B applied to a grades managing system. Let aspect B be responsible for giving bonuses and factors to grades, and let aspect A be in charge of enforcing a required grades format, by rounding non-integer grades and replacing all the grades above 100 by 100. Both aspects are applied before the publication of the grades, so the static parts of their pointcuts are the same. However, aspect A should be applied only if the grade to be published is not an integer or exceeds 100. Clearly, a computation might arrive at a place before grade publishing with an integer grade below 100, thus matched by the dynamic part of B's pointcut only (and not of A's), but as a result of B's modifications, a non-integer grade or a grade above 100 is obtained, bringing the computation to a state that is matched by A's pointcut as well.

Previously, two kinds of join-points have been examined: shadow join-points and actual join-points. A shadow join-point of aspect A, as mentioned above, is a place in the code of the base program that is matched by the static part of A's pointcut. An actual join-point of A is a state in a computation of the system at which the advice of A is actually applied. However, for the purpose of our analysis, a third type, *arrival join-points*, is needed:

Definition 2. *A state s in a computation of the woven system is called an* arrival join-point *of aspect A if s is the first state of a pointcut occurrence of A - the state when a join-point is first reached in that occurrence.*

Note that an arrival join-point differs from a shadow join-point because it is matched also by the dynamic part of A's pointcut descriptor. It also differs from an actual join-point: when the case of shared join-points was not possible, every arrival join-point reachable in the woven system became an actual join-point, but now it does not have to be so, because other aspects can intervene.

Note also that the pointcut of A identifies states either before or after some events of interest, but the definitions above are applicable for both cases. And if A is an `around` advice, it either has a `proceed`, and then can be viewed as a combination of two advice pieces - one before, and one after the corresponding event, or A has no `proceed`, and then can be viewed as a `before` advice, one of the effects of which is a change in the program counter of the base system. If indeed A's advice changes the program counter of the base system, the end of its execution is also the end of the current pointcut occurrence - both according to our intuition and to the definitions above.

Now let a state s be a join-point matched by aspect A, that appears inside pointcut occurrence π. We distinguish between four possible cases of other aspects' behavior that can influence the result of weaving A into a system:

1. Aspect B executed before A in π changes a value of some variable used by A as an input to its computations.
2. Aspect C executed after A in π changes a value of some variable updated by a computation of A.
3. Aspect D executed before A in π brings the system to a state s' which is not a join-point of A any more.
4. Aspect E executed after A in π invalidates the condition on which the join-point predicate depended, thus removing a join-point of A after A has already been executed at it.

The following analysis enables us to determine whether the above described influences actually cause an interference: Let the semantics of A be given by the assumption-guarantee pair (P_A, R_A), where R_A is the guarantee of A that must hold in any woven system containing A, provided the system into which A has been woven satisfied the assumption of A, P_A. Note that A can be woven into a system that does not satisfy P_A, but then R_A is not guaranteed to hold in the resulting system. We denote by $V_{in}(A)$ a set of variables A uses as input to its computations, and by $V_{out}(A)$ - a set of variables in which A stores the result of its computations.

Case 1. **Change Before (CB).** In case an aspect B executed before A at s changes a value of a $v \in V_{in}(A)$, the result of A's calculations might differ from the one we would get if the value of v has not been changed from the moment the computation arrived at s till the moment the advice of A was executed. If the guarantee of A contains a requirement for A's correctness, and this requirement is formulated in terms of a specific connection between the value of v when we arrive at a join-point and the value of v after the computation of A is finished, R_A will be violated in this case. (This can happen, for instance, in variant (1) of the Logging and Encrypting example above: we anticipate that the message string written to the log is the one created by the user and readable by the user, but if Encrypting is executed before Logging, what we actually get in the log is the encrypted message, because the contents of the message was changed by the Encrypting aspect.) Note that if the requirement for correctness of A's calculations binds the values at the end of A's execution only to the values at the beginning of execution of A (as in variants (2) and (3) of the Logging and Encrypting example, with Encryption before Logging), it will not be violated in this case.

Case 2. **Change After (CA).** In case some aspect C executed after A at s changes a value of a $v \in V_{out}(A)$, the guarantee of A will be violated if it required preservation of the result of A's computation till some future point in the execution where the value of v is used. (As in variant (2) of the example, when Logging occurs before Encryption). Otherwise, as in variant (3) of the example with Logging before Encryption, the guarantee of A will not be influenced. If, indeed, a requirement for preservation of the value of v till some state *use_v* is part of A's guarantee, then part of A's assumption should be that in the base system the value of v is not modified from the actual place of application of A's advice till arrival to the *use_v* state.

Case 3. **Invalidation Before (IB).** In this case there is no state in A's pointcut occurrence at which A is executed. Such a situation happens, for example, with Logging and Authorization aspects from Section 4 when the Authorization aspect is applied before Logging and the authorization of the user fails, thus preventing message sending, and removing the join-point of the Logging aspect. In variant (4) of the Logging specification, this leads to violation of the guarantee of Logging, as a message was prepared for sending and should have been logged, but the Logging aspect never has a chance to be applied, because the authorization failure finishes the pointcut occurrence.

Case 4. **Invalidation After (IA).** In this case A is executed at some point at which it shouldn't have been applied, because when arriving to the point of interest, the weaver "does not know" that the reason for A's application will be removed by one of the aspects coming after A. If the specification of A requires that it is applied only if followed by some event in the future, and this following event is removed by another aspect, then the specification of A is violated. This is the case, for example, in variants (1), (2) and (3) if the Authorization aspect is applied after Logging and the authorization of the user fails. Note that in variant (4), on the other hand, the guarantee of Logging is not violated if Logging precedes Authorization.

3. SPECIFICATION OF ASPECTS WITH POSSIBLY SHARED JOIN-POINTS

3.1 Guided Specification Construction

In order to be able to detect situations in which application of other aspects at a common join-point may contradict the specification of the examined aspect, the specification of the aspect must be expressive enough. The LTL specifications include:

- "$\mathsf{G}\,\varphi$" ("Globally") - meaning that the formula φ is true from the current state on.
- "$\mathsf{F}\,\varphi$" ("Finally") - from the current state a state in which φ holds can be reached.
- "$\mathsf{O}\,\varphi$" ("Once") - dual to "Finally": a state satisfying φ occurred earlier in the computation.
- "$\varphi\ \mathsf{U}\ \psi$" ("Until") - a state in which ψ holds is reached later in the computation, and until then φ holds.
- "$\varphi\ \mathsf{W}\ \psi$" ("Weak Until") - almost like "Until", but a state in which ψ holds does not have to be reached. In this case φ holds from now on forever.

We say that a computation satisfies an LTL formula if this formula holds in its first state. From the analysis in Section 2 the need for the following predicates arises:

- *at(ptc)*: assuming that *ptc* is the predicate defining A's pointcut, the predicate $at(ptc)$ means that the computation has just arrived at a join-point of A. It is useful for reasoning about what happened in the computation after the moment it arrived at a possibly shared join-point. In fact, this is the predicate marking the arrival join-points of A.
- *after_prev_asp(A)*: this predicate becomes true at the moment the weaver has applied all the aspects that preceded A at the current shadow join-point, according to the algorithm of the current weaver. The usage of this predicate is twofold: First, the user has to refine the definition of A's pointcut by taking into consideration the new predicate, *after_prev_asp(A)*, because now A should only be applied at states satisfying both *ptc* and *after_prev_asp(A)*, so the pointcut of A becomes $ptc \wedge after\_prev\_asp(A)$ (which matches the definition of the set of all the actual join-points of A). Second, the predicate is used in assumptions added to A's specification when the cases of "change before", "change after" and "invalidation before" presented in Section 2 are possible, to be able to reason about the behavior of the base system from the moment its computation arrives at a join-point of A till the moment A's advice is actually executed.
- *promise_ful(A)*: this flag is used by the weaver in order to give each of the aspects sharing a shadow join-point exactly one chance to be applied at it, as is explained later in Section 3.2. The flag $promise\_ful(A)$ is false when the computation arrives at a shadow join-point, becomes true at the moment the execution of A's advice begins, and remains true until the computation leaves this shadow join-point.
- *asp_ret(A)*: this predicate describes the possible return states of the aspect. This is needed for some of the cases below. Typically, the aspect return state has the same control location as the join-point state (the values can change, but not the program counter of the join-point). For the Logging aspect, for example, the base state is actually not changed, and only the log (local to the aspect) is modified. However, it does not have to be so in general. Thus in order to define the *asp_ret(A)* predicate, the user is proposed a default predicate, automatically constructed by the system as described in [11]. The idea of construction is to create a system containing representations of all the possible computations of an aspect from all its possible initial states, without actually applying the aspect to any specific base system (this is done using the MAVEN tool [9] and some built-in functionality of the NuSMV [3] model-checker), and then to build a predicate describing all the states of this system that satisfy

the return conditions of the aspect. This default predicate can then be manually modified.

Using the above predicates, all the requirements mentioned in Section 2 can be expressed, though not all the predicates are needed for all the cases of specification : sometimes some of them can be abstracted out without loss of precision of modeling and of subsequent verification. Below we present a way to express each of the additional requirements.

The construction of the refined specification can be automatic, but user-guided: several guiding questions will be presented to the user, and the answers to these questions will determine the new requirements. The construction process will thus be as follows:

Step 1. Here we will treat the dependency of our aspect, A, on its input variables, in order to find out whether the values of the input variables need to be preserved between the arrival and the actual join-points of A (in order to be able to treat the "change before" case from Section 2). The user is asked the following question:

Q. 1: Are there any input variables of A for which the advice of A depends on the value as it is at the arrival join-point and not as it is when the advice of A actually starts its execution?

- If yes, the user should provide a list of variables for which such a dependency exists.
- For each variable v in the list, we add the following CB (for "Change Before") statement to the assumption of A:

  $$CB(v) = \mathsf{G}[(at(ptc) \wedge v = V) \rightarrow ((v = V \ \mathsf{W} \ (after\_prev\_asp(A) \wedge v = V))]$$

  where V is a logical variable keeping the value of v as it was at the arrival to the join-point.
- If there are no variables in the list, nothing is added to the specification of A at this step.

Step 2. Here we treat the case when part of the effect of the aspect is modification of some state variables, and this effect should be preserved till some point in the future of the computation. This is important for the "change after" case from Section 2. The questions asked here are:

Q. 2: Are there any state variables of the system into which A is woven the value of which should be preserved after A's execution is finished? (For example, variables modified by A, or variables that are semantically connected to A's local variables.)

- If yes, the user is asked to fill in a table with two columns: the first column is the name of the variable, v, and the second is a state predicate *use_v* describing the state of the woven system until which the value of v should be preserved. For example, for variant 2 of the Logging aspect, that logs messages as they are sent to the server, the message should not change between the moment it has been logged and the moment it is actually sent. Thus the *use_v* predicate will describe the moment of actual sending of the message (see Section 4 for more details). After the table is filled out, for each variable v with state predicate *use_v* in the table, we add the following CA (for "Change After") statement to the assumption of A:

  $$CA(v) = \mathsf{G}[(asp\_ret(A) \wedge v = V) \rightarrow (v = V \ \mathsf{W} \ (use\_v \wedge v = V))]$$

  where V is a logical variable keeping the value of v as it was at the end of the execution of A's advice.
- If there are no variables in the list, nothing is added to the specification of A at this step.

Step 3. In this step we construct requirements corresponding to the "invalidation before" case in Section 2. Before the problem of common join-points in modular verification was considered, there existed an implicit assumption that all the arrival join-points of an aspect are its actual join-points. But when a join-point might be shared, this is not necessarily so, because the join-point can be invalidated; thus an additional explicit assumption of this possibility is needed. The user is asked the following question:

Q. 3: Does it have to be that each time an arrival join-point of A is reached, A is eventually executed at it? That is, is it an error if previously executed aspects invalidate the condition for A's application?

- If no, nothing is added to the assumption of A in this step.
- If the answer was "yes", the following IB (for "Invalidation Before") statement is added to the assumption of A:

  $$IB \triangleq \mathsf{G}[at(ptc) \rightarrow (ptc \ \mathsf{W} \ (after\_prev\_asp(A) \wedge ptc))]$$

Step 4. The goal of this step is to enable the verification process to treat the case of "invalidation after" from Section 2. We ask the user the following questions:

Q. 4.1: Does the reason for a state to be A's join-point lie in the future of the computation? That is, does A's pointcut descriptor refer to any event following the join-point? For example, is the advice of A a "before" advice?

- If no, nothing is added to the assumption of A in this step.
- If the answer was "yes", the next question is asked:

Q. 4.2: Is it an error if the advice of A is performed, but the presumably-following event does not follow? (For example, because the future computation was changed by other aspects)

- If the answer is "no", nothing is added to A's assumption in this step.
- If the answer is "yes", the user is required to provide a state predicate, *foll_event*, meaning that the desired following event has just occurred. The user is then prompted to provide some optional restrictions on the values immediately after A's execution, the values at the moment the desired event occurs, and the connections between them (including, for example, value preservation). The restrictions should be given in the form of two predicates: *vals_after_asp* and *vals_at_foll_event*. The default value for both predicates is *true*.
- The following IA (for "Invalidation After") statement is then added to the assumption of A:

$$IA \triangleq \mathsf{G}[(asp\_ret(A) \wedge vals\_after\_asp) \rightarrow F(foll\_event \wedge vals\_at\_foll\_event)]$$

After the above automatic modifications, the specification constructed both captures the requirements of the user regarding the desired effect of aspect application, and contains sufficient assumptions to make the modular verification results applicable to systems with aspects sharing join-points.

3.2 Influence on aspect modeling

For the purpose of automatic modular verification of aspects ([9, 11]) and interference detection ([10]), the following corrections to the modeling process are performed in order to obtain a correct weaving of advice models into the generic representations of suitable base systems:

- As follows from the discussion in Step 3 of Section 3.1 (treating the case of "invalidation before"), the pointcut definition of A should be refined to be $ptc' = ptc \wedge after\_prev\_asp(A)$, so that ptc' marks actual and not arrival join-points of A, because only at these points the advice of A is now executed. This change of the aspect model is done automatically.
- In order to model an aspect with possibly shared join-points, we need to be able to model returning of the advice to the join-point from which its execution started, so that the same advice will not be applied again at this point, but the other aspects will be able to execute. When several aspects can share a join-point, the weaver has to give them all exactly one chance to be applied at it. This can be viewed as fulfilling a promise to each one of these aspects. Thus a flag *promise_ful(A)* is added to the variables of the weaver for each aspect A.

3.3 Full Specification and Verification Process

Given a library of aspects, two things are important for its usage: correctness of each aspect alone with respect to its assume-guarantee specification, and interference detection among the aspects. The question of possibly shared join-points is already important when the specification of individual aspects is defined. At this stage one of the tools for detection of potential interference at common join-points detection can be run, e.g. [1], and only if a potential interference is detected the specification refinement described in Section 3.1 has to be performed for the potentially interfering aspects. (If no tool for potential interference detection is run, all the aspect specifications should undergo the process from Section 3.1, to ensure the soundness of the verification process.)

After all the aspects in the library are specified and augmented as described above, existing tools for modular aspect verification ([9, 11]) and interference detection ([10]) should be run. Modularity of verification here means that the correctness of the individual aspects and of their combinations is verified independently of any concrete base system, thus dividing the whole verification process into two independent parts: whenever an aspect, or a collection of aspects, are to be actually woven into a base system, one part of verification is to check that the base system satisfies the assumptions of all the aspects given, and the other part is to ensure that all the aspects are correct w.r.t. their assume-guarantee specifications, and do not interfere. Such a modularity enables us to check the correctness and interference-freedom of the library of aspects off-line and once and for all, and not each time some aspects are actually woven into a given base system, thus the verification effort is very much reduced. Another advantage of modular verification is that the models verified are smaller, as we never need to actually examine a woven system, and this enhances the model-checking process (and sometimes even makes it possible).

4. EXAMPLES

We illustrate our analysis and verification approach on a collection of aspects that can be a part of a communication-aspects library. The aspects presented here are applicable for systems with message-passing, and they are variants of the aspects used as an example in [1]. They are also mentioned in Section 2.

Logging aspect (L) logs the message - sending in the system. As described earlier, there are four variants of the logging aspect in the library:

- L_1: Logging all the sent messages as the user originally attempted to send them.
- L_2: Logging all the messages that were actually sent to the server (the message is logged as it was sent).
- L_3: Logging the frequency of message sending.
- L_4: Logging all the attempts to send a message (the message is logged as it was originally attempted to be sent by the user).

Now we need to construct the specifications for the above variants of Logging aspect. The following predicates and variables definitions will be used in the construction:

- *msg_to_send*: a predicate which is true when a message is about to be sent, that is, when message sending is attempted. That is the moment before the message-sending procedure is actually called, and the parameters to the method call are represented by the variables

msg_c and *msg_t*, containing the two parts of the message to be sent: the contents and the creation time, respectively.

- *msg_send*: a predicate which is true at the moment a message (with contents *msg_c* and creation time *msg_t*) is sent.
- *in_log* ($< str >$) : a predicate that is true if the string "str" appears in the log.

The pointcut of all the variants of the aspect is the moment before the message-sending procedure is called. More formally, $ptc = msg\_to\_send$. The guarantees of the aspects emerge in the usual way from the purpose of each of them, and are written more formally below. If there would be no possibility of sharing join-points, the assumption of all the logging aspect variants would be that if a message is sent, there indeed was an attempt to send this message (i.e., this very same message has been passed as a parameter to the message-sending procedure). More formally,

$$P_L \triangleq \mathsf{G}([msg\_send \wedge msg\_c = C \wedge msg\_t = T] \rightarrow \mathsf{O}[msg\_to\_send \wedge msg\_c = C \wedge msg\_t = T])$$

However, we are aware of the possibility of each of the aspects to share a join-point with other aspects, such as Encryption and Authorizatoin, thus additional assumptions for the aspects are constructed according to the procedure from Section 3.1.

Specification for L_1*:*. A possible guarantee for L_1 is:

$$R_{L1} \triangleq \mathsf{G}([at(msg\_to\_send) \wedge msg\_c = C \wedge msg\_t = T \wedge F(msg\_send)] \leftrightarrow [\mathsf{F}(in\_log(< X, T >))])$$

meaning that messages that appear in the log are all the sent messages, but as they were first attempted to be sent by the user. (Note that the fact that each message is accompanied by creation time information ensures a one-to-one correspondence between messages and lines in the log.)

The answers for the assumption-construction questions for L_1 are as follows:

Q.1: "Yes". The aspect depends on the contents and time information of the message as they were at the join-point, thus the values of the variables *msg_c* and *msg_t* should be preserved. Thus, substituting into the template $CB(v)$, the following statements are added to the assumption of L_1:

$$CB(c) = \mathsf{G}[(at(msg\_to\_send) \wedge msg\_c = C) \rightarrow ((msg\_c = C) \mathsf{W} (after\_prev\_asp(L1) \wedge msg\_c = C))]$$

and

$$CB(t) = \mathsf{G}[(at(msg\_to\_send) \wedge msg\_t = T) \rightarrow ((msg\_t = T) \mathsf{W} (after\_prev\_asp(L1) \wedge msg\_t = T))]$$

Q.2: "Yes". The time information of the message should be kept intact till the moment the message is actually sent. There is one entry in the table: the variable *msg_t*, matched by the *msg_send* predicate. Thus the addition to the aspect assumption at this stage is:

$$CA(t) = \mathsf{G}[(asp\_ret(L_1) \wedge msg\_t = T) \rightarrow ((msg\_t = T) \mathsf{W} (msg\_send \wedge msg\_t = T))]$$

Q.3: "No". If the message will not be sent, it should not appear in the log, thus the advice of L_1 should not be applied for it. Nothing is added to the assumption of L_1 at this stage.

Q.4.1: "Yes". The advice of L_1 is a "before" advice.

Q.4.2: "Yes". It is an error if a message that is not sent and will not be sent appears in the log. The desired following event is the event of sending the message (defined by its creation time only, as that is what matters for the purpose of L_1). Thus $foll\_event = msg\_send$, $vals\_after\_asp = (msg\_t = T)$ and $vals\_at\_foll\_event = (msg\_t = T)$. Substituting into the IA template, we obtain the following addition to L_1's assumption:

$$IA = \mathsf{G}[(asp\_ret(L_1) \wedge msg\_t = T) \rightarrow F(msg\_send \wedge msg\_t = T)]$$

Specification for L_2*:*. A possible guarantee for L_2 is:

$$R_{L2} \triangleq \mathsf{G}([\mathsf{F}(msg\_send \wedge msg\_t = T \wedge msg\_c = C)] \leftrightarrow [\mathsf{F}(in\_log(< C, T >))])$$

meaning that a message appears in the log if and only if it has been, or will be, sent.

The construction of the assumption for L_2 is performed similarly to that of L_1, with only two differences: An additional variable, *msg_c*, should be preserved after the aspect finishes its computation (affecting the $CA(v)$ and IA statements), and no values from arrival join-point should be kept (making $CB(v)$ *true*). Together we obtain that the addition to the assumption of L_2 as a result of the guided specification construction procedure consists of the following statements:

$$CA(c) = \mathsf{G}[(asp\_ret(L_2) \wedge msg\_c = C) \rightarrow ((msg\_c = C) \mathsf{W} (msg\_send \wedge msg\_c = C))]$$

$$CA(t) = \mathsf{G}[(asp\_ret(L_2) \wedge msg\_t = T) \rightarrow ((msg\_t = T) \mathsf{W} (msg\_send \wedge msg\_t = T))]$$

and

$$IA = \mathsf{G}[(asp\_ret(L_2) \wedge msg\_c = C \wedge msg\_t = T) \rightarrow F(msg\_send \wedge msg\_c = C \wedge msg\_t = T)]$$

Specification for L_3*:*. A possible guarantee for L_3 is:

$$R_{L3} = \mathsf{G}([\mathsf{F}(msg\_send \wedge msg\_t = T)] \leftrightarrow [\mathsf{F}(in\_log(< T >))])$$

meaning that the log contains all the creation-times of the sent messages.

The construction of the assumption for L_3 is almost the same as for L_2, except for the fact that the value of *msg_ts_c* need not be preserved after the aspect finishes its computation (thus giving the same $CA(t)$ and IA statements as for L_1). Thus the addition to the assumption of L_3 is

$$CA(t) = \mathsf{G}[(asp\_ret(L_3) \wedge msg\_t = T) \rightarrow ((msg\_t = T)\ \mathsf{W}\ (msg\_send \wedge msg\_t = T))]$$

and

$$IA = \mathsf{G}[(asp\_ret(L_3) \wedge msg\_t = T) \rightarrow F(msg\_send \wedge msg\_t = T)]$$

Specification for L_4:. A possible guarantee for L_4 is:

$$R_{L4} \triangleq \mathsf{G}([at(msg\_to\_send) \wedge msg\_c = C \wedge msg\_t = T] \leftrightarrow [\mathsf{F}(in\_log(< C, T >))])$$

meaning that the log contains exactly the messages attempted to be sent by the user.

The construction of the assumption for L_4 is almost the same as for L_1, with the following differences only:

- The answers to Question 2.1 and Question 4.1 are negative, as the logged message does not have to be sent, so $CA = true$ and $IA = true$ in this case.
- The answer to Question 3 is positive, as all the message sending attempts should be logged, including those aborted because of authorization failure.

Thus the additions to the assumption of L_4 are:

$$CB(c) = \mathsf{G}[(at(msg\_to\_send) \wedge msg\_c = C) \rightarrow ((msg\_c = C)\ \mathsf{W}\ (after\_prev\_asp(L_4) \wedge msg\_c = C))]$$

$$CB(t) = \mathsf{G}[(at(msg\_to\_send) \wedge msg\_t = T) \rightarrow ((msg\_t = T)\ \mathsf{W}\ (after\_prev\_asp(L_4) \wedge msg\_t = T))]$$

and

$$IB = \mathsf{G}[at(msg\_to\_send) \rightarrow (msg\_to\_send\ \mathsf{W}\ (after\_prev\_asp(L4) \wedge msg\_to\_send))]$$

Encrypting aspect (E) is responsible for encrypting messages before sending. E should guarantee that each time a message is sent, it is encrypted. In fact, there is more to E: each time a message is received, it is decrypted. But this part is irrelevant to our example, so we'll ignore it here. E's guarantee can be written as:

$$R_E \triangleq \mathsf{G}(msg\_send \rightarrow encrypted(msg\_c))$$

where the predicate *encrypted(msg_c)* means that the contents of the sent message are encrypted. The assumption of E, constructed by the procedure in Section 3.1, emerges from the fact that the encrypted message value should be preserved till (and if) it is actually sent:

$$P_E = CA(msg\_c) = \mathsf{G}[(asp\_ret(E) \wedge msg\_c = C) \rightarrow (msg\_c = C\ \mathsf{W}\ (msg\_send \wedge msg\_c = C))]$$

Authorization aspect (A) ensures that a message is sent to the server only if the current user has the needed permissions to communicate with the server. A's guarantee can be

$$R_A \triangleq \mathsf{G}(msg\_send \rightarrow permit\_usr\_send)$$

where the predicate *permit_usr_send* means that the user has enough permissions to send the message. When constructing the assumption of A, all the answers to the questions asked happen to be negative, thus A does not need to assume anything about the base system, and we can take

$$P_A \triangleq true$$

After the aspects are specified as above, if the usual verification procedure is applied, several cases of interference will be detected, as shown in Figure 1. A cell in the table corresponding to aspects pair $< M; N >$ can have one of the following values:

- "—" means that there is no interference when weaving first M and then N into any appropriate system;
- "X" - if the check is irrelevant, for example, we do not check interference among the aspect and itself. In our case we also do not check interference between different variants of the logging aspect, because we assume that only one of these aspects is woven into a system each time.
- Otherwise, there is interference among the two aspects if M is woven before N, and the cause of the interference is written in the cell, according to the classification from Section 2: "CB" stands for Change Before, "CA" - for Change After, "IB" - for Invalidation Before, and "IA" - for Invalidation After.

first \ second	E	A	L1	L2	L3	L4
E	X	---	CB	---	---	CB
A	---	X	---	---	---	IB
L1	---	IA	X	X	X	X
L2	CA	IA	X	X	X	X
L3	---	IA	X	X	X	X
L4	---	---	X	X	X	X

Figure 1: Interference checks summary.

For example, the cell $< E, L_1 >$ is marked by CB, meaning that the Encryption aspect, if woven first, invalidates the assumption of the first variant of Logging, and that the violated part of the assumption is related to the "change before" case from Section 2: changing parameter values between arrival and actual join-points of L_1. And the cell

$< L_1, A >$ is marked by IA as the Authorization aspect, when woven after the first variant of Logging, invalidates the part of Logging specification related to the "Invalidation After" case from Section 2: removing a join-point of the aspect after its advice has already been applied.

5. CONCLUSIONS

This paper has concentrated on a problematic issue of aspect semantics: the possible interference that can arise from shared join-points. As an aid to programmers, an interactive semi-automatic augmentation of the specification is suggested. The questions asked and the results of formal verification should help the user understand the fine points of such interactions, and how they could affect the correctness of their aspect systems.

6. REFERENCES

[1] M. Akşit, A. Rensink, and T. Staijen. A graph-transformation-based simulation approach for analysing aspect interference on shared join points. In *AOSD*, pages 39–50, 2009.

[2] L. Bergmans. Towards detection of semantic conflicts between crosscutting concerns. In *AAOS Workshop at ECOOP'03*, 2003.

[3] A. Cimatti, E.M. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: a new Symbolic Model Verifier. In *CAV'99*, LNCS 1633, pages 495–499. Springer, 1999. `http://nusmv.itc.it`.

[4] Curtis Clifton and Gary T. Leavens. MiniMao$_1$: Investigating the semantics of proceed. *Science of Computer Programming*, 63(3):321374, 2006.

[5] Simplice Djoko Djoko, Rémi Douence, and Pascal Fradet. Aspects preserving properties. In *PEPM*, pages 135–145, 2008.

[6] R. Douence, P. Fradet, and M. Sudholt. Composition, reuse, and interaction analysis of stateful aspects. In *Proc. of 3th Intl. Conf. on Aspect-Oriented Software Development (AOSD'04)*, pages 141–150. ACM Press, 2004.

[7] Rémi Douence, Pascal Fradet, and Mario Südholt. A framework for the detection and resolution of aspect interactions. In *GPCE*, pages 173–188, 2002.

[8] Pascal Durr, Lodewijk Bergmans, and Mehmet Aksit. Reasoning about semantic conflicts between aspects. In *ADI'06*, pages 10–18, 2006.

[9] M. Goldman and S. Katz. MAVEN: Modular aspect verification. In *Proc. of TACAS 2007*, volume 4424 of *LNCS*, pages 308–322, 2007.

[10] E. Katz and S. Katz. Incremental analysis of interference among aspects. In *FOAL '08*, pages 29–38. ACM, 2008.

[11] E. Katz and S. Katz. Modular verification of strongly invasive aspects. In *Languages: From Formal to Natural*, volume 5533 of *LNCS*, pages 128–147. Springer, 2009.

[12] Günter Kniesel. Detection and resolution of weaving interactions. *T. Aspect-Oriented Software Development*, 5:135–186, 2009.

[13] I. Nagy, L. Bergmans, and M. Aksit. Declarative aspect composition. In *Software Engineering Properties of Languages and Aspect Technologies (SPLAT) Workshop*, 2004.

[14] István Nagy, Lodewijk Bergmans, and Mehmet Aksit. Composing aspects at shared join points. In *NODe/GSEM*, pages 19–38, 2005.

[15] Dong Ha Nguyen and Mario Südholt. VPA-based aspects: Better support for aop over protocols. In *4th IEEE International Conference on Software Engineering and Formal Methods (SEFM'06)*, pages 167–176, 2006.

[16] M. Wand, G. Kiczales, and C. Dutchyn. A semantics for advice and dynamic join points in aspect-oriented programming. *T. on Programming Languages and Systems*, 26(5):890–910, 2004.

[17] N. Weston, F. Taiani, and A. Rashid. Interaction analysis for fault-tolerance in aspect-oriented programming. In *MeMoT'07*, pages 95–102, 2007.

Rewriting Logic Model of Compositional Abstraction of Aspect-Oriented Software

Yasuyuki Tahara
The University of Electro-Communications, Tokyo, Japan
tahara@is.uec.ac.jp

Akihiko Ohsuga
The University of Electro-Communications, Tokyo, Japan
akihiko@ohsuga.is.uec.ac.jp

Shinichi Honiden
National Institute of Informatics and The University of Tokyo, Japan
honiden@nii.ac.jp

ABSTRACT

Abstraction is an operation of software specifications widely used in formal development and verification. One of the desirable features of the operation is compositionality. It would make abstraction easier to deal with if a system can be abstracted by composing the individual abstractions of the components of the original system. It is considered that compositional abstractions of aspect-oriented software would be useful because the base system and the aspects can be individually abstracted. However, there are only a few research results dealing with these operations consistently because the relation between abstraction and aspect weaving is logically too complicated. This paper proposes a formal model to solve the difficulty of compositional abstraction of aspect-oriented software. Our model is based on an enhanced version of the equational abstraction approach in rewriting logic that is an algebraic specification framework. We first validate our model by applying it to an example of state machine and next describe our approach to compositional abstraction.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques—*object-oriented design methods*; D.2.4 [**Software Engineering**]: Software/Program Verification—*formal methods*; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*algebraic approaches to semantics*

General Terms

Design, Theory, Verification

Keywords

Abstraction, State Machines, Rewriting Logic, Compositionality

1. INTRODUCTION

Aspect-oriented software development (AOSD) is spreading quickly and widely these days. It is considered as providing a modularization facility to large-scale practical programs from a direction different from the traditional approaches such as structured, object-oriented, and component-based ones. In AOSD, cross-cutting concerns are extracted from the specifications of both the static structures and the dynamic behaviors and are encapsulated as aspects. In this paper, we focus on behavioral specifications. The behaviors of aspect-oriented software are so complicated that it is not easy to inspect such software manually if it has a practical scale and complexity. This is because formal models and formal verification techniques for aspect-oriented software are desired and many approaches [9, 1, 10, 20, 22, 21, 5, 13, 12, 11] are proposed.

The desirable features of formal models of modularization approaches include facilities of modular or *compositional reasoning and verification*. We mean by the word "compositional" that if we check some property for each module individually we can conclude that the entire system satisfies the same or another property. In this paper, we focus on the abstraction relation between behavioral specifications used in efficient model checking, which is a verification technique. Abstraction means the existence of a mapping between the state spaces of two state-transition behavior models that preserves the transition relations and some properties satisfied by the states. The model of the target of the mapping is called the abstract model. The significance of this relation comes from the following fact. If an abstract model satisfies a property written in some temporal logic, the refined model also satisfies the property. From this viewpoint, any existing approaches are insufficient. Only Jagadeesan et al. [11] deal with the (bi)simulation relation between specifications that is relevant to abstraction in a compositional manner. However, it is not easy to see if two given specifications have the relation or not.

In this paper, we propose a formal model of aspect-oriented software based on an algebraic specification framework called *rewriting logic* [14]. We also present an approach to specify abstraction relations on the basis of the notion of *equational abstraction* [15]. Rewriting logic is a logical framework in which we can derive equality and rewriting relations between terms from a set of axioms called a *rewrite theory* consisting of equations and rewrite rules. As a computation model, a state of a system is represented by an equivalence class of terms with respect to the equality, and transitions are represented by rewriting relations between the terms. Equational

abstraction is a very simple framework to create an abstract model by adding a set of equations. The projection function from the terms representing the system state to their equivalence classes is the abstraction mapping. In our approach, the behaviors of the base system and the aspect are individually modeled by the rewriting relations of the state terms of each system in one rewrite theory. We can create the rewrite theory that models the system in which the aspects are woven into the base system. Then the union of the sets of equations between the base systems and the aspects, respectively, can produce an abstraction mapping between the entire systems in a consistent way. As a result, for example, our approach enables us to carry out abstraction of an aspect-oriented system specifications by abstracting the base system and the aspect specifications individually as shown in Figure 1. This feature would be especially useful for aspect-oriented software because the entire specifications tend to become much larger and much more complicated than the base system and the aspect specifications. Our approach would reduce the cost of abstracting the entire system to the total of the costs of abstracting each components.

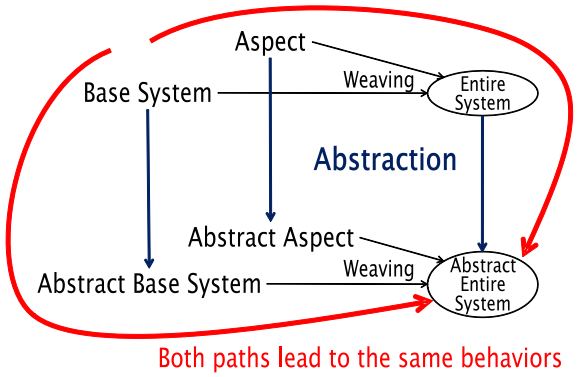


Figure 1: Individual Abstraction of Base System and Aspect

This paper is organized as follows. In Section 2 we summarize the background information of our approach consisting of the state machine model of aspects for compositional model checking and rewriting logic. Section 3 presents our aspect modeling method and justify of our model by demonstrating that we can model an existing formal model correctly. Section 4 describes the equational abstraction in our approach and shows that our model can realize compositional abstraction. Section 5 compares our proposal with the related research efforts. Section 6 presents some concluding remarks and future work.

2. BACKGROUNDS

In this section, we summarize the background information of our approach consisting of the state machine model of aspects for compositional model checking and rewriting logic.

2.1 State Machine Model of Aspects

A state machine M is a tuple $(S, S_0, \rightarrow, L)$ consisting of the set of the states S, the set of the initial states $S_0 \subseteq S$, the transition relation $\rightarrow \subseteq S \times S$, and the labeling function $L : S \rightarrow 2^{AP}$ where AP is the finite[1] set of atomic propositions. We assume that different truth values can be assigned to each atomic proposition in AP at different states in S. For any state s, $L(s)$ is the set of atomic propositions that are true at s. This definition of a state machine corresponds to a Kripke structure in the literature [7] enhanced with the initial states. "$\rightarrow$" must be total, that is, $\forall s \in S\, \exists s' \in S\, (s \rightarrow s')$. For general binary relation $\rightarrow \subseteq S \times S$ that is not total, we can create a total relation $\rightarrow^{\bullet} = \rightarrow \cup \{(s,s) | s \in S, \neg \exists s'\, (s \rightarrow s')\}$ that is an extension of $\rightarrow$.

The following definitions form a simplified version of Katz and Katz [12].

DEFINITION 1. *An* aspect machine *A over a set of atomic propositions AP is a tuple $(S_A, S_0^A, S_{ret}^A, \rightarrow_A, L_A)$ where S_A is the set of the states, $S_0^A \subseteq S_A$ is the set of the initial states, $\rightarrow_A \subseteq S_A \times S_A$ is the transition relation, $L_A : S_A \rightarrow 2^{AP}$ is the labeling function, and $S_{ret}^A \subseteq S_A$ is the set of return states such that $\forall s \in S_{ret}^A \forall s' \in S_A\, (s \rightarrow s'$ implies $s = s')$.*

DEFINITION 2. *A* pointcut descriptor *ρ over a set of atomic propositions AP is a predicate on finite sequences of labels. This means that $\rho(\lambda)$ is a boolean value for each $\lambda = l_0 l_1 \ldots l_n$ where $l_i \subseteq AP\ (i = 0, \ldots, n)$.*

DEFINITION 3. *For a state machine $M = (S, S_0, \rightarrow, L)$ and a (finite or infinite) state sequence $s_0 s_1 \ldots (s_n)$, $label(s_0 s_1 \ldots (s_n))$ is a label sequence $L(s_0)L(s_1)\ldots(L(s_n))$. A pointcut descriptor ρ* matches *a finite state sequence $s_0 s_1 \ldots s_n$ if and only if $\rho(label(s_0 s_1 \ldots s_n))$ is true .*

DEFINITION 4. *Let $B = (S_B, S_0^B, \rightarrow_B, L_B)$ be a state machine over a set of atomic propositions AP_B, ρ be a pointcut descriptor over AP_B, and "pointcut" be a symbol that is not an element of AP_B. Another state machine $B^\rho = (S_{B^\rho}, S_0^{B^\rho}, \rightarrow_{B^\rho}, L_{B^\rho})$ is said to be a* pointcut-ready machine *for B and ρ if and only if the following conditions hold.*

- $S_{B^\rho} \supseteq S_B$
- $L_{B^\rho} : S_{B^\rho} \rightarrow 2^{(AP_B \cup \{pointcut\})}$
- $\forall s_0, \ldots, s_k \in B^\rho\, (s_0 \rightarrow_{B^\rho} s_1 \rightarrow_{B^\rho} \ldots \rightarrow_{B^\rho} s_k$ *and* $s_0 \in S_0^{B^\rho}$ *implies* $(\rho(label(s_0 s_1 \ldots s_k))$ *if and only if pointcut* $\in L_{B^\rho}(s_k)))$
- $\forall l \in (2^{AP_B})^\omega\, ((\exists \pi_{B^\rho} :$ *path in* $B^\rho\, (label(\pi_{B^\rho}) = l))$ *if and only if* $(\exists \pi_B :$ *path in* $B\, (label(\pi_B) = l)))$

  where for a set S, S^ω is the set of infinite sequences of elements of S.

DEFINITION 5. *Suppose the following constructs are given.*

- *An aspect machine $A = (S_A, S_0^A, S_{ret}^A, \rightarrow_A, L_A)$ over AP*
- *A pointcut descriptor ρ over AP*
- *A state machine $B = (S_B, S_0^B, \rightarrow_B, L_B)$ called a* base machine *over $AP_B \supseteq AP$ and its pointcut-ready machine $B^\rho = (S_{B^\rho}, S_0^{B^\rho}, \rightarrow_{B^\rho}, L_{B^\rho})$*

[1] AP may be an infinite set in general. However, we deal with only finite sets as AP in order to enable M to be represented in rewriting logic.

Then we define as follows the augmented machine $\tilde{B} = (S_{\tilde{B}}, S_0^{\tilde{B}}, \rightarrow_{\tilde{B}}, L_{\tilde{B}})$ *in which the aspect machine is woven into the base machine.*

- $S_{\tilde{B}} = S_{B^\rho} \cup S_A$
- $S_0^{\tilde{B}} = S_0^{B^\rho}$
- $\rightarrow_{\tilde{B}} = \{(s,t) \in \rightarrow_{B^\rho} | \mathit{pointcut} \notin L_{B^\rho}(s)\} \cup \rightarrow_A$
  $\cup \{(s,t) \in S_{B^\rho} \times S_0^A | \mathit{pointcut} \in L_{B^\rho}(s),$
  $L_{B^\rho}(s) \cap AP = L_A(t)\}$
  $\cup \{(s,t) \in S_{ret}^A \times S_{B^\rho} | L_A(s) = L_{B^\rho}(t) \cap AP\}$
- $L_{\tilde{B}} = L_{B^\rho} \cup L_A$ *as sets of pairs*

Note that we omitted the treatment of fair states because we do not mention model checking in this paper.

2.2 Rewriting Logic

Next we explain rewriting logic. It can be summarized as follows.

- The most primitive construct of rewriting logic is a *term* that is a syntactic representation of a data or a state. Each term may have *sorts* representing data types.
- A logical formula of rewriting logic is an *equality* relation or a *rewriting* relation between terms. An equivalence class of terms represents a state of a system. A transition between states is represented by the rewriting relation between equivalence classes induced from the one between terms. For example, suppose $[t_1]$ and $[t_2]$ represent two states where $[t]$ denote the equivalence class t belongs to. Then a rewriting relation between terms $t_1 \rightarrow t_2$ induces $[t_1] \rightarrow [t_2]$ representing a transition from $[t_1]$ to $[t_2]$.

A term is composed by symbols for constants, variables, and operators that represent primitive data, placeholders for terms used to express generic equations or rewrite rules, and data structure constructors or operations on data, respectively. A constant symbol is usually treated as an operator symbol with no arguments. For example, $f(a,x)$ is a term if a is a constant symbol, x is a variable symbol, and f is an operator symbol with two arguments. Mixfix operators such as "$+ - \times /$" for numbers can be treated by the placeholder symbol "_". For example, $\_ + \_(2,3)$ can also be written as $2+3$ by replacing the two "_"s with the two arguments 2 and 3. A term is said to be *closed* if it includes no variable symbols.

Rewriting logic usually deals with sorts representing data types. It can be decided if a term *has* a sort or not. If a term t has a sort s, we call t a term of the sort s. Although it is recently usual to assign multiple sorts to one term at the same time, we do not deal with such cases in this paper only for simplicity. However, it would be not difficult to extend our approach to such general cases. The assignment of sorts to terms is derived from the initial assignment of sort information to the variable and the operator symbols. As for the example of the term $f(a,x)$ above, if a and x has the sorts s_1 and s_2 respectively and f is defined as an operator producing a term of the sort s from two arguments with the sorts s_1 and s_2, $f(a,x)$ has the sort s. We write these definitions as $a : s_1$, $x : s_2$, and $f : s_1 \times s_2 \rightarrow s$. We also write $\mathrm{Term}(s)$ for the set of the terms having the sort s.

The logical formulae of rewriting logic are equality relations or rewriting relations between terms. They respectively are expressed by the symbols "=" and "$\rightarrow$" and derived from axioms of equality and rewriting relations according to some inference rules. Each type of axioms is called equations and rewrite rules respectively. A (conditional) equation is a logical formula "$t = t'$ if $t_1 = t_1'$, ..., $t_n = t_n'$" where t, t', t_i, and $t_i' (i = 1, \ldots, n)$ are terms. A (conditional) rewrite rule is a formula "$t \rightarrow t'$ if $t_1 \Rightarrow t_1'$, ..., $t_n \Rightarrow t_n'$" where t etc. are terms in the similar way and $\Rightarrow$ denotes $=$ (the equality symbol) or $\rightarrow$ (the rewriting relation symbol). The both sides of each "=" and "$\rightarrow$" need to have the same sort. The inference rules are described as follows.

Reflexivity: For any term t, $t \Rightarrow t$

Symmetry: For any two terms t, t', if $t = t'$, $t' = t$

Congruence: For any operator f, $t_1 \Rightarrow t_1'$, ..., and $t_n \Rightarrow t_n'$ altogether imply $f(t_1, \ldots, t_n) \Rightarrow f(t_1', \ldots t_n')$, where all $\Rightarrow$'s coincide ($=$ or $\rightarrow$).

This rule expresses that each subterm can be rewritten individually.

Replacement: For any axiom "$t(x_1, \ldots, x_n) \Rightarrow t'(x_1, \ldots, x_n)$ if $s_1 \Rightarrow s_1', \ldots, s_n \Rightarrow s_n'$",

$s_1(\bar{w}/\bar{x}) \Rightarrow s_1'(\bar{w}/\bar{x})$, ..., and $s_n(\bar{w}/\bar{x}) \Rightarrow s_n'(\bar{w}/\bar{x})$ altogether imply $t(\bar{w}/\bar{x}) \Rightarrow t'(\bar{w}/\bar{x})$, where $\bar{w}$ denotes a sequence of terms $w_1, \ldots, w_n$ and $\bar{w}/\bar{x}$ denotes the componentwise substitution of x_i's to w_i's.

This rule produces relations instantiated from the axiom by substituting the variables $\bar{x}$ to terms $\bar{w}$ if w_i's satisfy the conditions.

Note that the original Replacement rule in the literature such as [14] follows from the above rule, the Congruence rule, and the Transitivity rule.

Equality and Transitivity: $t_1 \Rightarrow t_2$ and $t_2 \Rightarrow t_3$ altogether imply $t_1 \Rightarrow t_3$, where all $\Rightarrow$'s coincide (Transitivity), or one of the $\Rightarrow$'s in the premise is $=$ and the other two are $\rightarrow$'s (Equality). Note that the original Equality rule is derived by applying the above Equality rule twice for both sides of $\rightarrow$.

An axiomatic system of rewriting logic is called a *rewrite theory*. A rewrite theory is a tuple (S, Σ, V, E, R) where each component represents the set of sorts, the signature (the initial assignments of the operator symbols to their sort information), the initial assignments of the variable symbols to the sorts, the set of equations, and the set of rewrite rules, respectively.

In order to describe concrete rewrite theories, we use the notation of the Maude language [8] in which sorts, signatures, variable symbols, equations, and rewrite rules are declared with the keywords `sort`, `op`, `var`, `eq`, and `rl`[2], or their plural forms such as `sorts`, respectively. The following descriptions illustrate an example of rewrite theory.

[2]We also use `eq` and `rl` for conditional axioms instead of `ceq` and `crl` used in the literature for simplicity.

```
sorts S1, S2 .
vars x, y : S1 .
op c : -> S1 .
op f : S1 -> S2 .
op g : S1 S2 -> S2 .
eq g(c, f(c)) = f(c) .
rl g(x, f(y)) -> f(y) if f(x) = f(c) .
```

Then we explain the relationship between the state machine model and rewriting logic below as the basis of equational abstraction.

In fact, a rewriting relation of rewriting logic does not exactly correspond to a transition of a state machine model because of the Transitivity inference rule allowing compositions of rewriting relations. Therefore we need a precise counterpart concept for a transition in rewriting logic. We define a one-step rewriting relation $t \rightarrow^1 t'$ as a rewriting relation limited to one of the following cases.

- The last inference rule used in deriving the relation is either Equality or Replacement. The condition part of Replacement may contain any rewriting relations.
- The last inference rule used in deriving the relation is Congruence where only one assumption $t_i \rightarrow t_i'$ is one-step and all the others consist of identical terms ($t_j = t_j'$ for any $j \neq i$).

For a sort s, we write $\rightarrow^1 \cap (\mathrm{Term}(s) \times \mathrm{Term}(s))$ as $\rightarrow^1_s$.

We can create a state machine from a rewrite theory as follows. Let $\mathcal{R}$ be a rewrite theory including the following information.

- The following theory `BOOL`:

  ```
  sort Bool .
  ops true, false : -> Bool .
  op not_ : Bool -> Bool .
  op _and_ : Bool Bool -> Bool .
  ```

  as well as the equations about boolean algebras. For example,

  ```
  var X : Bool .
  eq not(false) = true .
  eq X and false = false .
  ```

- The specifications of the initial states and the labeling operator:

  ```
  sorts State, AP .
  op init : State -> Bool .
  op _|=_ : State AP -> Bool .
  ```

  where `State` and `_|=_` can be replaced with any sort and any similar operator respectively. For a term representing a state s, `init(`s`) = true` means that s represents an initial state.

In addition, we assume that there are only a finite number of closed terms for the sorts `State` and `AP`. Then the state machine $\mathcal{K}(\mathcal{R}, \texttt{State}, \texttt{\_|=\_})$ is defined as $(\mathrm{Term}(\texttt{State})/=, S_0, (\rightarrow^1_{\texttt{State}})^\bullet, L_{\texttt{\_|=\_}})$, where:

- $S_0 = \{[s]$ (the equivalence class to which the term s belongs) $|\, s$: closed term of the sort `State` and `init(`s`) = true`$\}$.
- $L_{\texttt{\_|=\_}}([s]) = \{p$: closed term of the sort `AP` $\mid$ `(` s `|=` p `) = true`$\}$.

3. ASPECT MODEL IN REWRITING LOGIC

We define the model of aspect-oriented behavioral specifications in rewriting logic in the following direction.

- We first define a *behavioral specification rewrite theory (BSRT)* to model behavioral specifications in general. A BSRT treats the behaviors as evolutions of terms called *configurations* by the rewriting relations. A configuration consists of an environment and a continuation. An *environment* is an assignment of values to the variables of the specifications (not the variable symbols of the rewrite theory) at each moment. A *continuation* is an expression of a behavioral specification representing the behaviors to be executed just after the moment.

  For example, suppose `(x : 0)` and `(x = 1)` are terms representing an environment and a continuation respectively. The former term means that the value `0` is assigned to the variable `x`. Note that `x` is treated as a constant symbol in rewriting logic as well as `0` and `1`. The latter denotes the behavior that stores the value `1` to `x` and terminates. Then the configuration `config((x : 0), (x = 1))` is rewritten to `config((x : 1), end)` in which `end` denotes the termination of the behaviors.

- We then define an *aspectual rewrite theory (ART)* as a BSRT specifying the base system and the aspects. The behavioral specifications of the base system and the advices are separately defined in one ART.
- An ART consists of terms called *augmented configurations*. An augmented configuration represents a state of the entire system in which the aspects are woven into the base system. Each augmented configuration consists of the following three elements: (1) the current system state, (2) the current continuation of the entire system, and (3) the data indicating either the aspect whose advice is currently executed or the fact that the base system is currently executed. The part of the current continuation of the entire system may be a behavioral specification of the base system or one of the advices.
- The actual specifications of the entire system in which the aspects are woven are given by an extension of the ART, called *augmented ART*. The extended part consists of rewrite rules. Some of them specify the behaviors at the beginnings of and at the ends of the advice executions. The remaining rules are those transformed from the specifications of the base system and the advice behaviors.
- An ART defines the join points by specifying the starting point of an advice execution and the point at which the advice execution finishes and the control returns to the base system. Many aspect-oriented languages including AspectJ specifies join points by pointcuts and advice types (before, after, around, etc.).
- The system specified by an ART can have multiple aspects. However, we assume that the system can execute only one advice at a time. Thus we do not deal with aspect compositions. On the other hand, multiple aspects may be woven at the same join point. Our

model assume that the order of weaving is nondeterminisitic. This may lead to some type of the aspect interference issues [16]. We assume these restrictions to make our model as simple as possible and able to deal with the approach of [12]. As described in Section 6, it would be possible to relax this restriction so as to, for example, make the model deal with aspect compositions.

- In order to represent the dynamic join point model, an ART specifies the conditions to detect the join points by boolean-valued functions over a list of augmented configurations representing an execution trace.

We define BSRTs as follows.

DEFINITION 6. *A rewrite theory $\mathcal{R}$ satisfying the following conditions is called a BSRT.*

- $\mathcal{R}$ protects `BOOL`. The word *protects* means that $\mathcal{R}$ add no other terms and no other equality relations than those inferred only from `BOOL`.
- $\mathcal{R}$ has the following sorts.
  - `ENV` for *environments*.
  - `BEH` for behavioral specifications.
  - `CONFIG` for *configurations*. A configuration here consists of a pair of an environment and a behavioral specification representing the current continuation.
- $\mathcal{R}$ has the operator `config : ENV BEH -> CONFIG` that is the only operator producing a term of the sort `CONFIG`. In addition, $\mathcal{R}$ is `CONFIG`-encapsulated [15], meaning that `CONFIG` only appears as the codomain of a single operator (in this case, `config` as shown above) and does not appear as an argument in any operator in $\mathcal{R}$.

As the opposite direction of creating $\mathcal{K}(\mathcal{R}, \texttt{State}, \_\models\_)$ from $\mathcal{R}$, we can create a rewrite theory $\mathcal{R}(M)$ from a state machine $M = (S, S_0, \rightarrow, L)$ as follows.

- $\mathcal{R}(M)$ include `BOOL`.
- $\mathcal{R}(M)$ include the following constructs.

```
sorts MState, AP, APS, Env, BEH, CONFIG .
var S, S' : MState .
var P : AP .
op initState : MState -> Bool .
op member : AP APS -> Bool .
op lbl : MState -> APS .
op d : -> Env .
op beh : MState -> BEH .
op config : Env BEH  -> CONFIG .
op _|=_ : CONFIG AP -> Bool .
op trans : MState MState -> Bool .
eq (config(d, beh(S)) |= P) = member(P, lbl(S)) .
rl config(d, beh(S)) -> config(d, beh(S'))
  if trans(S, S') = true .
```

where

  - The newly introduced sorts represent the states of the machine (`MState`) and the sets of atomic propositions (`APS`), respectively. We use only meaningless values (usually only one constant `d`) as environments. This is because the actual states are stored in the second argument of `config`. However, we can also use a meaningful sort or even `MState` as the environment sort instead of `Env`. If we use `MState`, the environment and the behavior part of the configuration changes simultaneously like `config(S, beh(S)) -> config(S', beh(S'))`.
  - The operators represent the initial state predicate (`initState`), the membership function for `APS` (`member`), a representative dummy environment (`d`), the system configuration constructor (`config`), the labeling function (`_|=_`), and the transition relation (`trans`) respectively.
- For each state $s \in S$, each atomic proposition p, and each set of atomic propositions ps, constant symbols (operator symbols with no arguments) `op` s `: -> MState`, `op` p `: -> AP`, and `op` ps `: -> APS`, respectively, are included in $\mathcal{R}(M)$.
- For each atomic proposition p and each set of atomic propositions ps, the equation `eq member(`p, ps`) = true` or `= false` for the same left-hand side (LHS) according to the membership relation, is included in $\mathcal{R}(M)$. Note that we need only a finite number of these equations because the number of atomic propositions, and therefore the number of the sets of them, are finite.
- For each state $s \in S$ and each set of atomic propositions ps, the equation `eq lbl(`s`) =` ps is included in $\mathcal{R}(M)$, if $L(s) = ps$.
- For each transition $s \rightarrow s'$, an equation `eq trans(`s, s'`) = true .` is included in $\mathcal{R}(M)$.

It is straightforward to see that $\mathcal{R}(M)$ is a BSRT and $\mathcal{K}(\mathcal{R}(M), \texttt{CONFIG}, \texttt{\_|=\_})$ is equivalent to M.

Next, we define ARTs as a specific type of BSRTs.

DEFINITION 7. *An ART is a BSRT satisfying the following conditions.*

- An ART has the following sorts.
  - `ASP` for aspects and a constant indicating the base system.
  - `AC` for augmented system configurations. An augmented system configuration consists of a tuple of the terms of the sorts `ENV`, `BEH`, and `ASP`, respectively.
  - `LAC` for lists of augmented system configurations. In detail, $\mathcal{R}$ includes the operations `op nil : -> LAC` and `op [_|_] : AC LAC -> LAC`. A list is treated as an execution trace used to judge the point in which an aspect is woven.
  - `TRC` for encapsulated terms of the sort `LAC`.
- An ART also has the following operators.

– `base : -> ASP` is the constant indicating that the base system is currently executed when it is used in an augmented configuration.

– `isBase : ASP -> Bool` is the predicate that becomes true if and only if the argument is `base`. If this is false, the argument is treated as an actual aspect. Therefore such a term must not be equal to `base`.

– `adv : ASP -> BEH` produces the behavioral specification of the advice included in the aspect.

– `as : LAC ASP -> Bool` is a predicate that becomes true in the following two cases. In the first case, the second argument is an aspect whose advice can be started immediately after the base system execution represented by the trace that is the first argument. In the second case, the second argument is `base` and no advices should be started immediately. `as` stands for "aspect selection".

– `rtn : CONFIG -> Bool` is a predicate that becomes true when the argument is a state in which the advice execution finishes and the system returns to the base system execution (`rtn` stands for "return").

– `rstrt : LAC BEH -> Bool` becomes true if and only if the second argument represents a base system continuation after the advice execution is finished (`rstrt` stands for "restart").

– `ac : ENV BEH ASP -> AC` is the only operator producing the terms of the sort `AC`.

– `trc : LAC -> TRC` is the only operator producing the terms of the sort `TRC`.

- Any term t of the sort `CONFIG` satisfying `rtn(`t`) = true` cannot be rewritten without using the Equality inference rule. This means that the advice cannot be executed beyond the point to return to the base system.

- There are no equations and no rewrite rules of the terms of the sorts `AC` and `LAC`.

We can add the constructs of the behavioral specifications needed to the entire system in which the aspect is woven into the base systems.

DEFINITION 8. *Let $\mathcal{R}$ be an ART. We define $\mathcal{R}^+$ as a rewrite theory in which the following constructs are added to $\mathcal{R}$.*

- The following specifications.

```
var E : ENV .
vars B, B' : BEH .
var A : ASP .
rl trc(L)
    -> trc([ac(E, B', base) | L])
  if L = [ac(E, B, A) | _],
     isBase(A) = false,
     rtn(config(E, B)) = true,
     rstrt(L, B') = true .
rl trc(L)
    -> trc([ac(E, adv(A), A) | L])
  if L = [ac(E, B, base) | _],
     as(L, A) = true,
     isBase(A) = false .
```

- The following rewrite rules for each rewrite rule "`config(`e, b`) -> config(`e', b'`) if` c" (c is the sequence of conditions) in $\mathcal{R}$:

```
rl trc(L) -> trc([ac(e', b', base) | L])
     if L = [ac(e, b, base) | _],
     as(L, base) = true, c .
rl trc(L) -> trc([ac(e', b', A) | L])
     if L = [ac(e, b, A) | _],
     isBase(A) = false, rtn(config(e, b)) = false,
     c .
```

In this definition, the system behaviors are represented by the rewriting relation between execution traces encapsulated by the operator `trc`. However, because each rewriting step only adds a new term of the sort `AC` to the head of the list (representing the last of the trace), the step can be seen as a rewriting step for the augmented system configuration. The aim of dealing with the traces is the detection of the join points.

The four different types of the rewriting relations described in the above definition represent the following behavior types respectively: (1) restarting the base system behavior execution immediately after the advice execution is finished, (2) starting to execute the advice, (3) continuing the base system execution, and (4) continuing the advice execution.

The theorem below justifies our model with respect to the state machine model via the construction of the rewrite theory $\mathcal{R}(M)$ from a state machine M.

DEFINITION 9. *Suppose B be a state machine, ρ be a pointcut descriptor, both of which are over a set of atomic propositions AP_B, and A be an aspect machine over $AP \subseteq AP_B$. We define an ART $\mathcal{A}(B, \rho, A)$ by adding the needed specifications to $\mathcal{R}(B)$. The details are presented in Appendix A.*

THEOREM 10. *In addition to the assumptions of Definition 9, suppose B_0^ρ be a pointcut-ready machine for B and ρ. Then we have a pointcut-ready machine B^ρ and the augmented machine $\tilde{B}$ for B^ρ and A satisfying the following condition if we write $\tilde{\tilde{B}}$ for $\mathcal{K}(\mathcal{A}(B, \rho, A)^+, \mathtt{TRC}, \_| = \_)$.*

$\forall l \in (2^{AP_B})^\omega$ $((\exists \pi_{\tilde{B}} :$ path in $\tilde{B}$ $(label(\pi_{\tilde{B}}) = l))$ if and only if $(\exists \pi_{\tilde{\tilde{B}}} :$ path in $\tilde{\tilde{B}}$ $(label(\pi_{\tilde{\tilde{B}}}) = l)))$

An outline of the proof is given in Appendix B.

4. EQUATIONAL ABSTRACTION IN ASPECT MODEL

Meseguer et al. proposed the equational abstraction approach as a model of abstraction for efficient model checking of rewriting logic specifications. This notion of abstraction is given in [6].

If we have a state machine $M = (S_M, S_0^M \rightarrow_M, L_M)$ and an equivalence relation $\equiv$ on S_M, we can create a *quotient state machine* $M/\equiv = (S_{M/\equiv}, S_0^{M/\equiv}, \rightarrow_{M/\equiv}, L_{M/\equiv})$ by the following definitions. We write $[s] \in S_M/\equiv$ as the equivalence class of $s \in S_M$.

- $S_0^{M/\equiv} = S_0^M / \equiv$
- For $s, s' \in S_M$, $[s] \rightarrow_{M/\equiv} [s']$ if and only if there exist $s_0 \in [s]$ and $s_0' \in [s']$ satisfying $s_0 \rightarrow_M s_0'$.
- $L_{M/\equiv}([s]) = \cap_{s_0 \in [s]} L_M(s)$

We say that $\equiv$ is *strict* if $s \equiv s' \in S_M$ implies $L_M(s) = L_M(s')$. The projection mapping $[\cdot] : S_M \rightarrow S_M/\equiv$ is called an *abstraction mapping* from M to $M/\equiv$. An abstraction mapping in the sense of [6] is a strict one. In this case, the satisfaction relation of some temporal logic is preserved by $M/\equiv$, if we define the satisfaction relation in the usual way presented in [6]. If $M' = (S_{M'}, \rightarrow_{M'}, L_{M'})$ is a state machine isomorphic to $M/\equiv$, that is, there is a bijection between A' and $S_M/\equiv$ preserving the transition relations and the labeling function, we also say M' is an abstract structure of M.

Then we describe the notion of equational abstraction that is a simplified version of [15].

THEOREM 11. *Let $\mathcal{R} = (S, \Sigma, V, E, R)$ be a rewrite theory including the specifications needed to create $\mathcal{K}(\mathcal{R}, \texttt{State}, \_| = \_)$. In addition, let E' be a set of conditional equations of the terms of the sort* `State` *and $\mathcal{R} \cup E' = (S, \Sigma, V, E \cup E', R)$. We define an equivalence relation $\equiv_{E'}$ on Term(*`State`*)/ $=_{\mathcal{R}}$ by*

$[t] \equiv_{\mathcal{R}} [t']$ if and only if $t = t'$ in $\mathcal{R} \cup E'$

Then, if the following conditions hold, $\mathcal{K}(\mathcal{R}, \texttt{State}, \_| = \_)/\equiv_{\mathcal{R}}$ is equivalent to $\mathcal{K}(\mathcal{R} \cup E', \texttt{State}, \_| = \_)$.

- *$\mathcal{R}$ is* `State`*-deadlock free, that is, $(\rightarrow^1_{\texttt{State}})^\bullet = \rightarrow^1_{\texttt{State}}$. In other words, there is at least one one-step rewriting starting from any term of the sort* `State`*.*
- *$\mathcal{R}$ is* `State`*-encapsulated.*
- *$\mathcal{R} \cup E'$ protects* `BOOL`*.*

Then we apply equational abstraction to our aspect model. Suppose that $\mathcal{R}$ is an ART with the sort of the atomic propositions `AP` and the satisfaction relation predicate `op _|=_ : TRC AP -> Bool`. It is clear that $\mathcal{R}^+$ and $\mathcal{R}^+ \cup E'$ is `TRC`-encapsulated. Therefore, if $\mathcal{R}^+$ is `TRC`-deadlock free, E' leads to an abstraction of $\mathcal{K}(\mathcal{R}^+, \texttt{TRC}, \_| = \_)$.

The compositionality of the equational abstraction of our aspect model is expressed by the following fact.

THEOREM 12. *$\mathcal{R}^+ \cup E'$ and $(\mathcal{R} \cup E')^+$ are the same.*

The proof is straightforward by observing that augmentation from $\mathcal{R}$ to $\mathcal{R}^+$ does not affect the equations.

Let $\equiv$ be the equivalence relation induced by E'. If we create Term(`TRC`)/ $\equiv$ and $\rightarrow^1_{\texttt{TRC}}$ from $\mathcal{R}^+ \cup E'$, we can complete a state machine for the abstraction of the entire system by adding a labeling operator `_|=_`. Term(`CONFIG`)/ $\equiv$ and $\rightarrow^1_{\texttt{CONFIG}}$ created from $\mathcal{R} \cup E'$ lead to a state machine including the abstract base system behaviors and the abstract advice behaviors separately. Therefore the coincidence of the two rewrite theories means the weaving and the abstraction operation are commutative. This fact represents the compositionality of abstraction in our approach.

5. RELATED WORK

There are many research efforts about formal behavior models of aspects [9, 1, 10, 20, 21, 5, 13, 12, 11, 4]. Some of them deal with some features of the approach of this paper. [13, 9, 12] treat finite state machine models of aspect-oriented systems mainly for the purpose of applying model checking. In addition, [9, 12] focus on compositional verification in which it is sufficient to verify the base system and the advice individually in order to verify the entire system. However, they only handle a single system at one time and do not consider relationships between systems including abstraction. [11] treats an aspect model based on the untyped lambda calculus. The main feature of this model is that it can model the (bi)simulation relation between two system expressions in a compositional way, that is, this relation is preserved under the weaving operation. However, as the literature admits, it is not easy to verify if two expressions have the (bi)simulation relation. Our model has limitations such as the first-order nature of algebraic specifications (while the untyped lambda calculus is higher-order) and our abstraction relation is a mapping. This enables us to create abstraction mappings easily. Although other formal models provide various viewpoints to aspect-oriented systems, they do not treat relationships between two systems either. [3] proposes an algebraic framework of feature-oriented development that may include AOSD. It also deals with stepwise refinement that can be considered as the inverse operation of abstraction. However, this paper does not discuss the formal correctness of the refinement. Recently, Braga [4] proposed an application of a formal framework called a constructive approach to modular structural operational semantics (constructive semantics) to aspect-oriented software. Although it does not deal with pointcuts depending on execution traces, it is promising to extend it with our approach to deal with traces.

There are also many researches about (semi-)automatic transformations of semiformal aspect behavior models mainly written in UML in the context of MDA (Model-Driven Architecture) [2, 23, 24, 17, 19, 18]. We can regard most of the transformations treated there as a generalization of the refinement relation in our approach. While we cannot discuss the correctness of the transformations in these approaches rigorously, they can treat practical situations with realistic scales and complexity. Therefore it is interesting to model them formally in our framework and evaluate the practical feasibility of our approach.

6. CONCLUSIONS

In this paper, we proposed a formal model of aspect-oriented systems based on rewriting logic and an approach of compositional equational abstraction for our model. Because our approach realizes a highly compositional way of establishing abstraction relation between aspect-oriented behavioral specifications, it is promising as a theoretical foundation of efficient AOSD methodologies.

In our approach, there are many limitations to be relaxed in the future. First, our model is too complicated and rather specific. The details of the configurations could be abstracted to the more general notion of states. Such abstraction would make our model much simpler. Our current model based on configurations could be obtained by refining the states back. The expressiveness of our approach is weak in comparison with the higher-order approaches. We also omitted the treatment of fair states. We are planning to treat aspect compositions by extending the aspect information in augmented configuration terms. We need to make

clear the limitations by trying to express various examples. Such trials will also enable us to evaluate the practical feasibility of our approach.

We need to apply our approach to concrete case studies to estimate how our approach can reduce the costs of reasoning about aspect-oriented systems. Such reasoning tasks include verification and model transformations.

Acknowledgments

This work was supported by the Ministry of Education, Culture, Sports, Science and Technology, Grant-in-Aid for Scientific Research (B). The authors would also like to thank the members of Ohsuga Lab., Honiden Lab., and GRACE center., especially Mr. Hiroyuki Nakagawa, Prof. Nobukazu Yoshioka, and Prof. Kenji Taguchi, for their enthusiastic discussions with us.

7. REFERENCES

[1] J. H. Andrews. Process-algebraic foundations of aspect-oriented programming. In *Proc. of Reflection '01*, pages 187–209. Springer-Verlag, 2001.

[2] U. Aßmann and A. Ludwig. Aspect weaving with graph rewriting. In *Proc. of GCSE '99*, pages 24–36, London, UK, 2000. Springer-Verlag.

[3] D. Batory. Feature-oriented programming and the AHEAD tool suite. In *Proc. of ICSE 2004*, pages 702–703, Washington, DC, USA, 2004. IEEE Computer Society.

[4] C. Braga. A constructive semantics for basic aspect constructs. In *Semantics and Algebraic Specification*, pages 106–120, 2009.

[5] G. Bruns, R. Jagadeesan, A. S. A. Jeffrey, and J. Riely. muABC: A minimal aspect calculus. In *Proc. Concur*, volume 3170 of *Lecture Notes in Computer Science*, pages 209–224. Springer-Verlag, 2004.

[6] E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. *ACM TOPLAS*, 16(5):1512–1542, September 1994.

[7] Edmund M. Clarke, Orna Grumberg, and A. Peled. *Model Checking*. MIT Press, 1999.

[8] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada. Maude: Specification and programming in rewriting logic. *Theoretical Computer Science*, 285(2):187–243, 2002.

[9] M. Goldman and S. Katz. MAVEN: Modular aspect verification. In *Proc. of TACAS'07*, pages 308–322, 2007.

[10] R. Jagadeesan, A. S. A. Jeffrey, and J. Riely. A calculus of untyped aspect-oriented programs. In *Proc. of ECOOP '03*, pages 415–427. Springer-Verlag, 2003.

[11] R. Jagadeesan, C. Pitcher, and J. Riely. Open bisimulation for aspects. In *Proc. of AOSD '07*, pages 107–120, New York, NY, USA, 2007. ACM.

[12] E. Katz and S. Katz. Modular verification of strongly invasive aspects. In *Languages: From Formal to Natural*, pages 128–147, 2009.

[13] S. Krishnamurthi and K. Fisler. Foundations of incremental aspect model-checking. *ACM Trans. Softw. Eng. Methodol.*, 16(2):7, 2007.

[14] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96:73–155, 1992.

[15] J. Meseguer, M. Palomino, and N. Martí-Oliet. Equational abstractions. *Theoretical Computer Science*, (403):239–264, 2008.

[16] I. Nagy, L. Bergmans, and M. Aksit. Composing aspects at shared join points. In *Proc. of Intl. Conf. on NetObjectDays*, pages 69–84. Springer, 2005.

[17] P. Sánchez, L. Fuentes, D. Stein, S. Hanenberg, and R. Unland. Aspect-oriented model weaving beyond model composition and model transformation. In *Proc. of MoDELS '08*, pages 766–781, Berlin, Heidelberg, 2008. Springer-Verlag.

[18] D. Simmonds, R. Reddy, R. France, S. Ghosh, and A. Solberg. An aspect oriented model driven framework. In *Proc. of EDOC '05*, pages 119–130, Washington, DC, USA, 2005. IEEE Computer Society.

[19] D. Stein, S. Hanenberg, and R. Unland. Expressing different conceptual models of join point selections in aspect-oriented design. In *Proc. of AOSD '06*, pages 15–26, New York, NY, USA, 2006. ACM.

[20] David Walker, Steve Zdancewic, and Jay Ligatti. A theory of aspects. In *Proc. ACM SIGPLAN International Conference on Functional Programming*, Uppsala, Sweden, 2003. ACM.

[21] M. Wand, G. Kiczales, and C. Dutchyn. A semantics for advice and dynamic join points in aspect-oriented programming. *ACM Trans. Program. Lang. Syst.*, 26(5):890–910, 2004.

[22] Mitchell Wand, Gregor Kiczales, and Christopher Dutchyn. A semantics for advice and dynamic join points in aspect-oriented programming. *TOPLAS*, 26(5):890–910, 2004.

[23] J. Whittle, A. Moreira, J. Araújo, P. Jayaraman, A. Elkhodary, and R. Rabbi. An expressive aspect composition language for UML state diagrams. In *Proc. of MoDELS '07*, pages 514–528. Springer, 2007.

[24] G. Zhang, M. Hölzl, and A. Knapp. Enhancing UML state machines with aspects. In *Proc. of MoDELS '07*, pages 529–543. Springer, 2007.

APPENDIX

A. DEFINITION OF $\mathcal{A}(B, \rho, A)$

We describe the definition again. Suppose B be a state machine, ρ be a pointcut descriptor, both of which are over a set of atomic propositions AP_B, and A be an aspect machine over $AP \subseteq AP_B$. We define an ART $\mathcal{A}(B, \rho, A)$ by adding the needed specifications to $\mathcal{R}(B)$.

- The constructs needed to an ART such as the specifications of `ASP` and `adv`.
- The following constructs corresponding to A.

```
sorts State', MSSeq .
var S, S1, S2 : State .
var S', S1', S2' : State' .
var M, M' : MSSeq .
var L : LAS .
var P : AP .
var A : ASP .
op beh' : State' -> BEH .
```

```
op lbl' : State' -> APS .
op asp : State' -> ASP .
op nilS : -> MSSeq .
op [[_|_]] : State MSSeq -> MSSeq .
op adl : State -> Bool .
op pc, pm : MSSeq -> Bool .
ops _==B_ : APS APS -> Bool .
op msSeq : LAS -> MSSeq .
op path : MSSeq -> Bool .
op _|=_ : TRC AP -> Bool .
op trans' : State' State' -> Bool .
eq isBase(asp(_)) = false .
eq adv(asp(S')) = beh'(S') .
eq msSeq([ac(_, beh'(S'), asp(_)) | L]) = nilS .
eq msSeq([ac(_, beh(S), base) | L]) = nilS
  if adl(S) = true .
eq msSeq([ac(_, beh(S), base) | L])
   = [[S | msSeq(L)]]
  if adl(S) = false .
eq path([[S]]) = initState(S) .
eq path([[S2 | M]]) = true
  if M = [[S1 | _]], trans(S1, S2) = true .
eq append([[]], M) = M .
eq append([[S | M]], M') = [[S | append(M, M')]] .
eq pc(M) = true
  if path(M') = true, pm(M') = true,
     M' = append(M, _) .
eq as([ac(_, _, base) | L], base)
   = not pc(msSeq(L)) .
eq as(L, asp(S')) =
    pc(msSeq(L)) and (lbl(S) ==B lbl'(S')) .
eq rstrt([ac(_, beh'(S'), _) | _], S)
    = (lbl(S) ==B lbl'(S')) .
eq trc([ac(_, beh(S), base) |_] |= P)
   = member(P, lbl(S)) .
eq trc([ac(_, beh'(S'), A) |_] |= P)
   = member(P, lbl'(S'))
  if isBase(A) = false .
rl config(d, beh'(S1')) -> config(d, beh(S2'))
  if trans(S1', S2') = true .
```

- For each state s of A, a constant symbol `op` s `: -> State'`.
- For each transition $s \to t$ of A, an equation `eq trans'(`s, t`) = true .`
- We provide additional states and transitions as follows to cope with the case of *strongly invasive aspects* in [12]. An aspect is said to be strongly invasive if it resumes to an unreachable state of the pointcut-ready machine.
  - For each label $P \subseteq AP$, a constant `op` s_{AP} `: -> State` representing an additional state and the specification of its label `eq lbl(`s_{AP}`) =` P.
  - For each additional state constant s_{AP} and a constant t of the sort `State`, an equation representing a transition between them `eq trans(`s_{AP}, t`) = true .`
- For each constant s of the sort `State`, an equation `eq adl(`s`) = true .` or `= false .` if s is an additional one or a state of B, respectively.
- The equations specifying the semantics of `pm` that satisfies `pm`(m) = `true` if and only if the pointcut descriptor ρ matches the inverse of the machine state sequence m[3].
- For each state s and each set of elements of AP ps, `eq lbl'(`s`) =` ps `.` if and only if $L'(s) = ps$. Note that ps can be represented by a constant of the sort `APS` with respect to AP_B because $AP \subseteq AP_B$.
- For each pair of constant symbols of the sort APS representing a set of atomic propositions $ps_1 \subseteq AP_B$ and $ps_2 \subseteq AP$, the equation `eq (`ps_1 `==B` ps_2`) = true .` or `= false .` according to if $ps_1 \cap AP = ps_2$ or not.
- For each state s of the aspect machine A, the equation `eq rtn(config(d, beh'(`s`))) = true .` or `= false .` according to if s is a return state ($s \in R_A$) or not.

Some constructs are explained in detail as follows.

- Because the initial state from which the aspect machine execution starts varies according to the last state of the base machine, we specify an aspect term other than `base` by encapsulating an aspect machine state representing the initial state with the operator `asp`. Thus we also specify `eq isBase(asp(_)) = false .` The advice extraction operator `adv` takes out the encapsulated state by `eq adv(asp(S')) = S' .`
- Although [12] treats only one aspect machine at a time, we can see that the framework of the literature implicitly deals with multiple aspects by the above observation. In addition, [12] specifies transitions between states of the base system and the advice that have the same label (the set of atomic propositions satisfied at a state). We can model these situations by adding the following constructs to the rewrite theory.
  - The sort `MSSeq` of sequences of the base machine states. Terms of this sort is used to detect the join points by checking them with the pointcut descriptor ρ. Such terms are extracted from the execution traces. Accordingly, we add the specification of the extraction operator `msSeq` and the pointcut predicate `pc : MSSeq -> Bool` that produces true if and only if the pointcut descriptor matches the state sequence.
  - Specifications of the equality operator `_==_` on the sort `APS` and `as`.
- For a machine state sequence m, `pc`(m) = true if and only if the inverse of m is a latter part $s_{i+1} \dots s_n$ of a path $\pi = s_0 s_1 \dots s_n$, includes only the states of B, s_i is not a state of B (that is, a state of A or an additional state), and is matched by ρ. As we show in the proof of the main theorem, we can consider the last state of the path as a pointcut state.
- `path` produces true if and only if its argument represents a finite path of B starting form an initial state.

B. PROOF OUTLINE OF THEOREM 10

First we construct B^ρ from B_0^ρ by adding the following constructs.

- For each label $P \subseteq AP$, an additional state s' satisfying $L_{B^\rho}(s') = P$.

[3]We assume that such algebraic specifications of ρ exists

- For each additional state s' introduced above and each state s of B^ρ, a transition $s' \to s$. The latter state may be an additional one or one originally in B_0^ρ.

Because all the additional states are unreachable in B^ρ, it is easy to see that B^ρ is also a pointcut-ready machine for B and ρ.

Next the transitions of $\tilde{\tilde{B}}$ are classified by the following lemma.

DEFINITION 13. *We write $e, s, a, l \rightrightarrows e', s', a', l'$ for a one-step rewriting relation in an augmented ART $\mathcal{R}^+$ `trc([ac(e, beh(s), a) | l] →`*
`trc([ac(e', beh(s'), a') | l'], or beh' instead of beh if its argument s or s' is a state of A.`

LEMMA 14. *A transition of $\tilde{\tilde{B}}$ is obtained from either one of the following four types of one-step rewriting relations.*

1. $\mathtt{d}, s, \mathtt{asp}(s'), l \rightrightarrows \mathtt{d}, t, \mathtt{base}, l'$ *for a return state s of A and a state t of B where $L_A(s) = L_B(t) \cap AP$ or an additional state t where $L_A(s) = L_B(t)$.*
2. $\mathtt{d}, s, \mathtt{base}, l \rightrightarrows \mathtt{d}, t, \mathtt{asp}(t), l'$ *for a state s of B and an initial state t of A where $L_B(s) \cap AP = L_A(t)$ and `pc(msSeq([[s|l]])) = true`.*
3. $\mathtt{d}, s, \mathtt{base}, l \rightrightarrows \mathtt{d}, t, \mathtt{base}, l'$ *for two states s and t of B where not `pc(msSeq([[s|l]])) = true`.*
4. $\mathtt{d}, s, \mathtt{base}, l \rightrightarrows \mathtt{d}, t, \mathtt{base}, l'$ *for an additional state s and a constant t of the sort `State`.*
5. $\mathtt{d}, s, \mathtt{asp}(s'), l \rightrightarrows \mathtt{d}, t, \mathtt{asp}(s'), l'$ *for two states s and t of A where s is not a return state.*

Proof: This lemma can be proven by the fact that a one-step rewriting relation of $\mathcal{A}(M, \rho, A)^+$ can be obtained only by applying either one type of the rewrite rule in Definition 8 and each corresponding pair of conditions are equivalent. □

Fix an $l \in (2^{AP_B})^\omega$. Because the proofs of the two directions of "if and only if" are almost symmetric, we show only the "only if" part below. Thus we also fix a path $\pi_{\tilde{B}}$ of $\tilde{B}$ where $\mathrm{label}(\pi_{\tilde{B}}) = l$ and try to create a path $\pi_{\tilde{\tilde{B}}}$ of $\tilde{\tilde{B}}$ with the same label.

LEMMA 15. *Let $\pi_{\tilde{B}}$ be a path satisfying the left-hand side. Then we can decompose this path into the fragments $\pi^0, \pi^1, \ldots (, \pi^n)$ satisfying either one of the following two conditions, where each fragment is a finite path except the last one π^n if it exists.*

1. *π^i starts from an state s of B^ρ ($s \in S_0^{B^\rho}$), ends with a pointcut state s' of B^ρ, that is, a state satisfying pointcut$\in L_{B^\rho}(s')$, if π^i is finite. Every other state is in S_{B^ρ} and not a pointcut state.*
2. *π^i starts from an initial state s of A ($s \in S_0^A$), ends with a return state s' of A, if π^i is finite. Every other state is in A and not a return state.*

Proof: It is straightforward to define π^i's by induction on i by taking the longest fragments satisfying the two conditions alternately. □

Let $l \in (2^{AP_B})^\omega$ be a label, $\pi_{\tilde{B}}$ be a path of $\tilde{B}$ satisfying $\mathrm{label}(\pi_{\tilde{B}}) = l$, and $\pi^0, \pi^1, \ldots (, \pi^n)$ be a decomposition of $\pi_{\tilde{B}}$ as in Lemma 15. We can create a path $\pi_{\tilde{\tilde{B}}}$ of $\tilde{\tilde{B}}$ by composing the fragments $\pi'^0, \pi'^1, \ldots (, \pi'^n)$ shown in the following lemma.

LEMMA 16. *Under the above assumptions, there is a sequence of path fragments $\pi'^0, \pi'^1, \ldots (, \pi'^n)$, where each fragment is finite except the last one π'^n if it exists, satisfying label(π^i) = label(π'^i) for each i, and the following two conditions.*

1. *If π^i satisfies the conditions 1 of Lemma 15, π'^i starts from `d`, s, `base`, l, where s is a state of B or an additional state. If π^i is finite, it also ends with `d`, s', `base`, l', where `pc(msSeq(l')) = true`. In addition, every other state is in S_B or an additional state constant.*
2. *If $\pi^i = s_0 s_1 \ldots (s_n)$ (s_n exists only if π^i is finite-length) satisfies the conditions 2 of Lemma 15, π'^i is `d`, s_0, `asp`(s_0), $l_0 \rightrightarrows$ `d`, s_1, `asp`(s_0), $l_1 \rightrightarrows$ `d`, s_2, `asp`(s_0), $l_2 \rightrightarrows \ldots (\rightrightarrows$ `d`, s_n, `asp`(s_0), $l_n)$.*

Proof: We can prove this by induction on i.

- If $\pi^0 = \pi$, this is an infinite path of B^ρ. By the definition of B^ρ, there is an infinite path of B with the same label as π. It is easy to obtain the desirable π'^0.
- If $\pi^0 = s_0 s_1 \ldots s_{i_0}$ is finite, it can be extended to an infinite path π_0^* because of the totality of $\to_{B^\rho}$. Then there is an infinite path of B^ρ with the same label as π_0^*. It is easy to obtain the desirable π'^0 by limiting the length of this path.
- To examine the cases for general i, we divide the following three cases: (1) π^i is infinite, (2) π^i is finite-length and satisfies the condition 1 of Lemma 15, and (3) π^i is finite-length and satisfies the condition 2 of Lemma 15. In case (1), we need not to proceed any more. In case (2), we can obtain π'^{i+1} by finding the next state of π'^i confirming the all the conditions of the case 2 of Lemma 14 and extending the path from it by connecting the transitions of the case 5 of the Lemma 14. In case (3), we need to be careful because π^{i+1} may start from an unreachable state. We explain this part of the proof in detail. Note that the last state s of π^{i+1} is reachable because it is a pointcut state by Lemma 15 and therefore there is a path that ends with s and is matched by ρ. Let s_0 be the first reachable state of π^{i+1} and π^+ be the path composed by a path from an initial state to s_0 and the latter part of π^{i+1} starting from s_0. By the totality of $\to_{B^\rho}$, we can extend π^+ to an infinite path π^{++} by adding transitions after s. By the definition of B^ρ, we have an infinite path π'^{++} of B such that label(π^{++}) = label(π'^{++}). Let the state of π'^{++} corresponding to s_0 and s be s'_0 and s, respectively, and $s'_0 s'_1 \ldots s$ be the subsequence of π'^{++}. If we also let $s'_{-k} s'_{-k+1} \ldots s'_{-1}$ be the sequence of additional state constant of $\mathcal{A}(B, \rho, A)$ that has the same label as $s_{-k} s_{-k+1} \ldots s_{-1}$ that is the initial segment of π^{i+1} before s_0, we have the following path of $\tilde{\tilde{B}}$: `d`, s'_{-k}, `base`, $l_0 \rightrightarrows$ `d`, s'_{-k+1}, `base`, $l_1 \rightrightarrows \ldots$ `d`, s'_{-1}, `base`, $l_{k-1} \rightrightarrows$ `d`, s'_0, `base`, $l_k \rightrightarrows$ `d`, s'_1, `base`, $l_{k+1} \rightrightarrows \ldots \rightrightarrows$ `d`, s', `base`, l, where the initial part until s'_0 consists of the additional transitions. It is straightforward to see that the execution trace makes the `pc` operator true because label(π^{++}) = label(π'^{++}) implies that ρ matches the both paths and the latter part of the trace is also a latter part of π'^{++}. □

Modeling Aspects by Category Theory

Serge P. Kovalyov
Technological Institute of Computer Technics
6 Institutskaya st. – 630090 Novosibirsk, Russia
+7-383-333-37-94
kovalyov@nsc.ru

ABSTRACT
A framework for formal analysis of aspect-oriented software development (AOSD) is proposed. AOSD is treated as enriching formal models of programs by traceable refinements that produce their systemic interfaces. Category-theoretic construction of architecture school is employed to formalize this approach. Aspect weaving and separation of concerns are defined as universal constructions. Aspect-oriented scenario modeling is discussed as an example.

Categories and Subject Descriptors
F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—Specification techniques

General Terms
Design, Theory

Keywords
Aspect Oriented Software Development, Architecture School, Traceability, Aspect Weaving

1. INTRODUCTION
Aspect oriented software development (AOSD) [2] aims at explicit separating and composing concerns elaborated in response to particular requirements. Concerns are usually much tangled and scattered across software modules due to intermingling and conflicting nature of requirements. Modularizing them or at least keeping them clearly distinguishable throughout the development process can drastically improve software maintainability. However, AOSD has gained lower level of adoption in software production than modular design up to now. As argued in [15], although the community agrees on what AOSD *is good for*, there are no common paradox-free understanding of what AOSD actually *is*. This is partly caused by lack of a sound uniform metamodel capable to elucidate handling of aspects within modular software development paradigms in a natural way.

We present an attempt to create such metamodel. It is based on the concept of tracing development process steps, since traceability is most compromised by tangling. Concerns are treated as sources of traceable refinements that eventually produce software artifacts. Attaching these refinements to program models allows identifying, composing (weaving), and separating concerns in the course of the development process. In order to provide formal semantics of these operations that doesn't rely on some specific development paradigm we employ category theory. It allows characterizing (mathematical) objects by their interrelations with other objects, avoiding appeal to their "interiors" (by which software artifacts created with different technologies much vary). Constructions in categories produce objects that satisfy extensional ("systemic") criteria: universality (existence and uniqueness of interrelation with similar objects), naturality (independence of multistep interrelation on the way it is traced), and so on. Such objects are usually determined uniquely up to an isomorphism, viz. appropriate abstraction of inessential difference. Visual diagrammatic notation is routinely used to specify constructions.

This motivates employing category theory as paradigm-independent formal tool to reason on software design. The very notion of modularity (the Holy Grail of AOSD) is captured as a class of diagrams that satisfy certain structural properties. Augmenting it with concepts of an interface and of a refinement (expressed in category-theoretic terms) leads to fundamental notion of an architecture school that provides uniform general representation of software design (see [4]). In this setting, we formally present enhancing a design technology by aspects as natural labeling of school constituents by concerns.

We employ scenario modeling as principal example of applying this construction to specific technology. On the one hand, it properly captures operational semantics of aspect-oriented programming that essentially consists in augmenting base program execution scenario with woven aspects. On the other hand, it is a widely used requirements engineering technique empowered with methods for transforming models to architectures, including those based on category theory [14]. Enhancing domain engineering of large-scale distributed systems with aspect-oriented capabilities facilitates rapid incremental non-invasive development [9].

2. ARCHITECTURE SCHOOLS
Category-theoretic approach to formalizing software systems design is being developed since 1970s. According to it, formal models (descriptions) of programs are represented as objects, and

actions of integrating (models of) individual components into (models of) systems are represented as morphisms. Composition of morphisms represents multistep integration; identity morphisms represent trivial integrating descriptions into themselves by "doing nothing". Resulting category of descriptions is denoted as *c-DESC*. An example can be found in the area of object-oriented design: classes (formally presented on UML) can be considered as objects in category-theoretic sense, with inheritance relations as morphisms. Another example, particularly pertaining to the topic of this paper, is modeling software systems by execution scenarios. The basic mathematical representation of a scenario is a partially ordered set (poset) [13]. Its elements are atomic events occurred during execution, partially ordered by causal dependence. Actions of scenarios integration are precisely homomorphisms of posets since neither events nor interactions shall be "forgotten" at integration. Hence category **Pos** of all posets and all their homomorphisms plays the role of *c-DESC*.

A system built from multiple components is represented as a *c-DESC*-diagram that consists of the components and their interconnections. Recall that a diagram is a functor of the kind $\Delta : X \to c\text{-}DESC$, where X is a small category (called a schema of Δ). A diagram can be "visualized" as a directed graph of a category X whose points are labeled by *c-DESC*-objects and arrows are labeled by *c-DESC*-morphisms. In order to facilitate category-theoretic reasoning on systems design decisions, we consider diagrams as objects of appropriate category (the (covariant) "super-comma category", see [11]). First, recall that all diagrams with schema X comprise a category, denoted $c\text{-}DESC^X$, whose morphisms are called natural transformations. Recall that a natural transformation from a diagram $\Delta : X \to c\text{-}DESC$ to a diagram $\Sigma : X \to c\text{-}DESC$ is a map $\varepsilon : \mathrm{Ob}\, X \to \mathrm{Mor}\, c\text{-}DESC$ that satisfies the following naturality condition. For every X-objects A, B, and every X-morphism $f : A \to B$ the equality $\Sigma(f) \circ \varepsilon_A = \varepsilon_B \circ \Delta(f)$ holds (in particular, we have $\varepsilon_A : \Delta(A) \to \Sigma(A)$ for every X-object A). A natural transformation ε can be visualized as a "prism" with graphs of Δ and Σ as bases, and arrows ε_A, $A \in \mathrm{Ob}\, X$, as lateral edges. Naturality condition ensures that composition induces minimal amount of auxiliary arrows that cross lateral faces of the prism, i.e. that component-wise integration of a system represented by Δ into a system represented by Σ established by ε respects every interconnection.

A notion of a natural transformation admits straightforward extension to diagrams with different schemas, by adding a functor that adjusts schemas. So a morphism from a diagram $\Delta : X \to c\text{-}DESC$ to a diagram $\Sigma : Y \to c\text{-}DESC$ is a pair $\langle \varepsilon, fd \rangle$ consisting of a functor $fd : X \to Y$ and a natural transformation $\varepsilon : \Delta \to \Sigma \circ fd$. If ε consists of identities (implying that $\Delta = \Sigma \circ fd$) and fd is injective, then Δ is called a subdiagram of Σ (graph of Δ is a labeled subgraph of graph of Σ).

Each object A forms a singleton diagram whose graph consists of the single point labeled by A. Morphisms between singleton *c-DESC*-diagrams are precisely *c-DESC*-morphisms between objects they constitute (in category theory it is said that $c\text{-}DESC^{\mathbf{1}}$, where **1** is a singleton category, is isomorphic to *c-DESC*). A morphism from an arbitrary diagram Δ to a singleton diagram is called a cocone. It can be visualized as a diagram in form of a "pyramid" with base Δ and edges directed from points of Δ to the distinct point called the vertex. A colimit of Δ, denoted colim Δ, is a cocone that is universal in a sense that every cocone δ over Δ factors through colim Δ uniquely (i.e. there exists a unique *c-DESC*-morphism c, called a colimit arrow, such that $\delta = \langle c, 1_{\mathbf{1}} \rangle \circ \mathrm{colim}\, \Delta$). Obviously a colimit is determined uniquely up to an isomorphism. Its vertex (called a colimit object) can be thought of as the least "container" that encapsulates all objects of Δ via edges respecting structure of their interconnections. For example, a colimit of a discrete diagram (i.e. the one whose schema has no morphisms except identities) is precisely a coproduct of its objects, which includes all of them (preserving their identity) and nothing more. Even an empty diagram may have a colimit, whose object is precisely an initial object (there exists exactly one morphism from it to any other object). An initial object represents a "componentless" system that can be uniquely integrated into every system (for example, "pure" integration middleware).

These considerations motivate employing a notion of a colimit as category-theoretic abstraction of system synthesis [10]. Existence of a colimit is a necessary condition for a *c-DESC*-diagram to represent a valid system. Clearly it is not sufficient, since various structural rules usually apply (e.g. type constraints in object-oriented design). Diagrams that actually produce systems are called well-formed configurations, and constitute class denoted as *Conf*. In scenario modeling, although any **Pos**-diagram has a colimit, a configuration is considered as well-formed only if its target system scenario is included into it explicitly, without employing any structural computations beyond constructing a coproduct. Such situation is common at requirements engineering where the analyst haven't yet collected enough information to establish powerful structural rules over requirement models. So the class *CPos* of all disjoint unions of **Pos**-cocones is used for *Conf*. Every discrete **Pos**-diagram is well-formed; it represents a disjoint union of parallel (non-interacting) scenarios. To see that "many" other diagrams are ill-formed, consider a pair of **Pos**-morphisms $\{a < b\} \leftarrow \{a, b\} \to \{a > b\}$, where arrows denote bijections $a \to a$, $b \to b$. This diagram exposes a and b as concurrent events, so it is impossible to include both conflicting orderings into the single scenario. In particular, the diagram's colimit object, which is a singleton poset, fails to represent integration result (as well as any other poset).

It is well known that integration capabilities of a component are completely determined by its specially devised "part" called an interface. Interfaces of formal models comprise a category denoted *SIG*; extracting an interface is expressed as a signature functor $sig : c\text{-}DESC \to SIG$. Although it shouldn't be injective (different components can have the same interface), it is required to be faithful, i.e. injective on each hom-set $\mathrm{Mor}(A, B)$ that consists of all *c-DESC*-morphisms with domain A and codomain B (otherwise it would fail to distinguish different ways to integrate a component A into a system B). Realizability of every interface is ensured via existence of a functor $sig^* : SIG \to c\text{-}DESC$ that produces a "default" implementation of every interface I. Specifically, $sig^*(I)$ has I as an interface (i.e. $sig \circ sig^* = 1_{SIG}$) and supports all integration capabilities of I (i.e. functor sig surjectively, hence bijectively, maps a hom-set $\mathrm{Mor}(sig^*(I), A)$ to $\mathrm{Mor}(I, sig(A))$ for every *c-DESC*-object A). In category-theoretic terms functor sig^* is called left adjoint to sig.

There exists a natural correlation between interfaces and configurations. First, conditions $\Delta \in Conf$ and $sig \circ \Delta = sig \circ \Sigma$ shall imply $\Sigma \in Conf$ for any *c-DESC*-diagrams Δ, Σ. This requirement establishes a kind of logical non-contradiction law for interfaces: no interface integration schema can be produced by both systems and illegal conglomerates of components. Second, interface extraction shall be natural with regard to composing systems: every colimit of *SIG*-diagram $sig \circ \Delta$ equals to $sig \circ \operatorname{colim} \Delta$ for each $\Delta \in Conf$. In other words, functor *sig* lifts colimits of diagrams from *Conf*. Naturality is actually two-fold: requirements enlisted above imply that functor *sig* preserves colimits of diagrams from *Conf*.

In scenario modeling, an interface of a scenario is a set of occurred events obtained by forgetting their order. Indeed, in order to identify an execution scenario of a component in a scenario of a system, it is necessary and sufficient to identify all events occurred within the component. So the canonical forgetful functor $|\text{–}| : \mathbf{Pos} \to \mathbf{Set}$, where **Set** is a category of all sets and all maps, is used for *sig*. It is easy to see that it satisfies all requirements above. In particular, default realization of an interface is represented by a functor $|\text{–}|^* : \mathbf{Set} \to \mathbf{Pos}$ that turns a set S into a discretely ordered poset $\langle S, = \rangle$ that seamlessly integrates into any scenario.

In addition to system composition, software development process contains steps of modeling individual components known as refinements. The process is commonly viewed as moving along two dimensions: horizontal structuring induced by "component-of" relationships and vertical structuring induced by "refined-by" relationships [10]. All descriptions and all refinements comprise a category, denoted *r-DESC*, with the same class of objects as *c-DESC*. A (trivial) example of a refinement is an isomorphism, so a subcategory of *c-DESC* that contains of all descriptions and all *c-DESC*-isomorphisms is required to be a subcategory of *r-DESC*. The naturality of refinement with regard to composing systems is imposed in the form that a collection of refinements of components constituting a system shall induce a refinement of the system. This can be formally expressed in terms of natural *r-DESC*-transformations, given that every discrete *c-DESC*-diagram is simultaneously an *r-DESC*-diagram. An arbitrary collection of refinements of objects of a *c-DESC*-diagram Δ is precisely a natural *r-DESC*-transformation $\varphi : |\Delta| \to \Sigma$, where $|\Delta|$ is discrete diagram that consists of all objects from Δ, and Σ is a discrete diagram consisting of all refinements results. If $\Delta \in Conf$, then a diagram $\Delta \oplus \varphi \in Conf$ shall exist, that has Σ as a subdiagram (and possibly extra points and arrows), and an *r-DESC*-morphism from a colimit object of Δ to a colimit object of $\Delta \oplus \varphi$.

In scenario modeling, refinement of a scenario consists in replacing atomic events with subscenarios in such a way that the order is fully inherited [5]. Formally, a refinement of a poset X to a poset A is identified with a surjective map $f : A \to X$ that satisfies the condition $\forall x \, \forall y \, f(x) \le f(y) \Leftrightarrow (x \le y \vee f(x) = f(y))$. Notice the swap of source and destination: as we will see below, it is a key to smooth aspect orientation. We will denote the category of all posets and all scenario refinements by *r-Pos*.

A tuple $\langle$*c-DESC*, *Conf*, *sig*, *r-DESC*$\rangle$, that satisfies all conditions enlisted above, is called an *architecture school*, and various examples of schools are considered, in [4]. Of particular interest are schools "over" **Set**, in which *c-DESC* is a concrete category over **Set** in the following sense. Descriptions are sets equipped with some structure (e.g. algebraic structures, topological spaces, etc), integration actions are maps that respect the structure, and *sig* is canonical functor $|\text{–}|$ that forgets the structure. Scenario modeling architecture school $SM = \langle \mathbf{Pos}, CPos, |\text{–}|, r\text{-}Pos \rangle$ is an example of an architecture school over **Set**.

3. ENHANCING DESIGN WITH ASPECTS

Our model of AOSD rests upon representing emergence of aspects in a software design technology as formal transformation of architecture schools. Indeed, AOSD can be generally considered as equipping software artifacts with certain labeling conveniently identifying concerns handled by their constituents. Original motivation of AOSD creators [7] stems from the fact that programming languages are too concise to allow tracing intermingled fragments of source code to their ultimate "goals". Different flavours of AOSD [2] greatly vary in labeling techniques (among which modularization is most welcome) but agree in pursuing transparent traceability, viz. ability to determine exactly what each fragment of a model is included into it for.

A metamodel of traceability proposed in [16] formally demonstrates that tracing is routinely compromised by refinement. A refinement may change the very "nature" of a model, e. g. when implementing a specification by means of a programming language. On the contrary, system composition is able to provide at least partial tracing back to components; difficulties arise at tracing concerns that *crosscut* boundaries of modular architecture (such as security). So ability to trace result of a refinement to its source means that reversing its direction (i.e. category-theoretic dualization) produces a *c-DESC*-morphism, called its *trace*. In order to preserve traceability in subsequent integration of a result into a system, a trace shall have right inverse at the level of interfaces. Indeed, if a refinement $r : X \to A$ satisfies a condition $sig(r^{\mathrm{op}}) \circ s = 1_{sig(X)}$ for some *SIG*-morphism $s : sig(X) \to sig(A)$, then *SIG*-morphism $sig(f) \circ s$ identifies $sig(X)$ in $sig(S)$ for every *c-DESC*-morphism $f : A \to S$.

Obvious example of traceable refinement is a *c-DESC*-isomorphism (recall that a dual to an isomorphism is identified with its inverse which is an isomorphism as well). Non-trivial traceable refinements are obtained as *r-DESC*-morphisms that coherently behave as duals to *c-DESC*-morphisms. Denote by *cr-DESC* the intersection of all such maximal common subcategories of *c-DESC* and *r-DESC*$^{\mathrm{op}}$ that contain all *c-DESC*-isomorphisms.

Definition 1. A *cr-DESC*-morphism t is called a *trace* provided that $sig(t)$ is a retraction (i.e. has right inverse). A *sig*-image of a trace is called a *labeling*. A dual to a trace is called a *traceable refinement*. □

In an architecture school over **Set** every labeling is a surjective map, so a traceable refinement $r : X \to A$ is a total antifunctional binary relation that is conservative with regard to structure. Its action can be described as expansion of points of $|X|$ to sets that comprise partitioning of $|A|$, projecting structural constraints defined on points of $|X|$ to some (possibly none) members of their expansion results. A point of $|X|$ can be considered as a concern that is elaborated by expansion, in accordance with intuitive notion of refinement. For example, in scenario modeling school *SM r-Pos*$^{\mathrm{op}}$ is a subcategory of **Pos**; every refinement is traceable, and literally determines a labeling of its target by points of its

source. Moreover, refinements allow tracing inclusions of components, viz. integration actions that leave inner structures of components intact. Inclusions are represented by regular **Pos**-monomorphisms; capability to trace them means that for every *r-Pos*-morphism $r : X \to A$ and inclusion $i : M \to X$ there exists an inclusion $i' : M \to A$ such that $r^{op} \circ i' = i$.

Observe that traceable refinements particularly tolerate configurations. Consider a diagram, called a *push* of Δ by φ, that consists of a diagram $\Delta \in$ *Conf* and a family φ of arrows directed from distinct points outside of Δ to points of Δ. Obviously a push has a colimit with the same object as Δ. Comprising φ from traces (so that φ^{op} is a natural *r-DESC*-transformation from $|\Delta|$ to a discrete subdiagram of a push), we see that traceable refinements are *non-invasive* with respect to system composition: if a push belongs to *Conf*, then it can be taken for $\Delta \oplus \varphi^{op}$, and appropriate isomorphism for a refinement of colimit objects. Non-invasive refinements are much appreciated within the context of AOSD. For example, this is obviously the case for scenario modeling school.

These considerations suggest that the AOSD objective can be achieved by equipping descriptions with traceable refinements that produce them, at least at the interface level. Such equipping is precisely the desired aspect labeling. We employ the construct of comma category (see [11]) to formalize it. We will work in specific comma category denoted as $sig \downarrow SIG$. Recall that its objects are pairs $\langle A, a : sig(A) \to X \rangle$, where $A \in$ Ob *c-DESC* and $a \in$ Mor *SIG*. A morphism from an object $\langle A_1, a_1 \rangle$ to an object $\langle A_2, a_2 \rangle$ is such pair $\langle f : A_1 \to A_2, b : \text{codom } a_1 \to \text{codom } a_2 \rangle$ that $b \circ a_1 = a_2 \circ sig(f)$. Denote by *AO* full subcategory of $sig \downarrow SIG$ whose objects are all pairs $\langle A, a \rangle$ in which a is a labeling.

In an architecture school over **Set** aspect labeling $a : |A| \to X$ of a description A consists in assigning each point of $|A|$ a point of set X that denotes the "name" of the aspect it belongs to. The labeling is essentially (up to an *AO*-isomorphism) an equivalence relation on $|A|$, equivalence classes representing individual aspects. Every such relation turns A into a valid aspect-oriented model, so aspects needn't respect its "modular" structure in any way. *AO*-morphisms are precisely such *c-DESC*-morphisms that preserve this additional equivalence relation. As we will see below, there exists a functor that turns *AO* into a concrete category over **Set**.

Objects of various categories that comprise *AO* can serve as interfaces of *AO*-descriptions, contributing to turning *AO* into full-scale architecture school. Specifically, the software designer have freedom to choose interfaces of aspect-oriented models to be either:

- original non-aspect-oriented models, obtained by functor *mod* that takes an *AO*-object $\langle A, a \rangle$ to a *c-DESC*-object A, for modular design tasks;
- aspect labelings, obtained by functor *asp* that takes $\langle A, a \rangle$ to a, for design and analysis of aspect structure;
- original model interfaces, obtained by functor $int = sig \circ mod$, for specification purposes.

Other options that refine (i.e. can be naturally transmuted to) original interfaces may be available in particular schools.

Refinements and well-formed configurations of aspect-oriented models are constructed by appropriate enrichment of modular "material". Let *tr-AO* be the subcategory of *AO* that consists of all *AO*-objects and all such *AO*-morphisms f that $mod(f)$ is a trace. Further, denote by *str* functor that takes $\langle A, a \rangle$ to codom a. Notice that, given an *AO*-diagram Δ, a diagram $|asp \circ \Delta|$ that consists of labelings of all objects of Δ can be viewed as a natural transformation of $int \circ \Delta$ to $str \circ \Delta$, i.e. $\gamma \circ \langle |asp \circ \Delta|, 1_{\text{dom } \Delta} \rangle$ is a cocone over $int \circ \Delta$ for each cocone γ over $str \circ \Delta$. Bearing this in mind, we will call a class *I-Dia* of *SIG*-diagrams *aspect-closed* if for any $\Sigma \in$ *I-Dia* an *AO*-diagram Δ satisfies the following conditions provided that $int \circ \Delta = \Sigma$:

- $mod \circ \Delta \in$ *Conf*;
- *SIG*-diagram $str \circ \Delta$ has a colimit;
- every colimit arrow c_Δ, such that $\text{colim } (str \circ \Delta) \circ \langle |asp \circ \Delta|, 1_{\text{dom } \Delta} \rangle = \langle c_\Delta, 1_{\mathbf{1}} \rangle \circ \text{colim } (int \circ \Delta)$, is a labeling;
- for every natural *tr-AO*-transformation $\varphi : \Sigma \to |\Delta|$ there exists an *AO*-diagram $\Delta \oplus \varphi$, such that Σ is its subdiagram, $int \circ (\Delta \oplus \varphi) \in$ *I-Dia*, and there exists a *tr-AO*-morphism $t : \langle C^{\oplus}, c_{\Delta \oplus \varphi} \rangle \to \langle C, c_\Delta \rangle$, where $C^{\oplus}$ is a colimit object of $mod \circ (\Delta \oplus \varphi)$ and C is a colimit object of $mod \circ \Delta$.

Denote by *AO-Int* the union of all aspect-closed classes of *SIG*-diagrams. It allows determining all configurations that retain modularization when constituent components gain labeling by aspects.

Definition 2. Given an architecture school $AR = \langle$*c-DESC*, *Conf*, *sig*, *r-DESC*$\rangle$, functor *ai* is said to *generate an aspect-oriented architecture school* (AO-school) from *AR*, if a tuple

$$AO_{ai}(AR) = \langle AO, \{\Delta \mid int \circ \Delta \in AO\text{-}Int\}, ai, tr\text{-}AO^{op} \rangle$$

is an architecture school, and there exists such functor *si* that $si \circ ai = int$. □

Theorem 3. Functors 1_{AO}, *mod*, *asp*, *int* generate AO-schools. □

The proof of the theorem consists in checking that $AO_{ai}(AR)$ satisfies all conditions for an architecture school whenever one of enlisted functors is taken for *ai*. In particular, functor *mod**, which is left adjoint to *mod* and defines inclusion of *c-DESC* into *AO*, takes a *c-DESC*-object A to an *AO*-object $\langle A, 1_{sig(A)} \rangle$. It represents the first step in enhancing a modular design technology by aspects: seed aspect structure coincides with an integration interface. *AO*-descriptions with non-trivial aspect structures emerge upon refining them in the course of AOSD process.

In scenario modeling school *SM*, aspects appear to be precisely labels that, being attached to elements, turn posets into *pomsets* [13]. Labels can be considered as event "names" denoting concerns they handle. Class *AO-Int* coincides with $|\text{–}| \circ$ *CPos*, which means that all configurations admit aspect orientation. Since every refinement is traceable, all of them are used at constructing *tr-AO*^{op}. Functor *mod** endows the discrete labeling on a scenario, equipping each event with a unique label (actually itself).

4. WEAVING AND SEPARATING ASPECTS

Elementary building blocks of aspect-oriented models are known as *aspects*. In an architecture school over **Set**, an aspect is precisely an *AO*-object whose *str*-image is a singleton set, i.e. a

terminal **Set**-object (there exists exactly one map from any other set to it). For example, an aspect in scenario modeling is precisely a pomset with all elements labeled with the same label. In order to generalize to arbitrary AO-school, observe that every morphism directed from a terminal object has left inverse (that typically is a trace, so aspects particularly tolerate tracing).

Definition 4. An *AO*-description A is called an *aspect* if $str(f)$ has left inverse for every *AO*-morphism $f : A \to X$. □

Proposition 5. If *c-DESC* has a terminal object **1**, then A is an aspect iff $str(A) = sig(\mathbf{1})$. □

Aspect-oriented program synthesis and decomposition techniques can be formalized as universal constructs in category *AO*. For example, weaving an *AO*-object W (advice) into an *AO*-object B (base) is represented as follows. Weaving rules determine join points in base B at which W is called through appropriate entry points. For example, a program written on an aspect-oriented extension of an object-oriented language, such as AspectJ [3], can be weaved to the base before/after method calls, access operations to fields, exception handlers, etc. In order to specify weaving rules, auxiliary *AO*-object C, called connector, is employed (see [12]) in a way that matching between entry points and join points is described as a pair of *AO*-morphisms $j : B \leftarrow C \rightarrow W : e$. Observe that morphism j is usually called a pointcut descriptor [3]. Weaver at first (virtually) produces enough copies of W, one for each join point, with appropriate entry point marked at each copy. Then binding entry points to matching join points establishes the weaving provided that it respects aspect structures of both models. In an architecture school over **Set** the first step of weaving can be formalized as constructing a product $C \times W$; subsequent binding of points is represented as appropriate pushout. These operations admit straightforward generalization to arbitrary school. Recall that a pushout is a colimit of a diagram that has a form of a pair of arrows with the same source. It is used in category theory to generalize set-theoretic operation of identifying "the same" elements in different sets.

Definition 6. An *aspect weaving* of a pair of *AO*-morphisms $j : B \leftarrow C \rightarrow W : e$, where B is called *base description*, W is called *description being weaved*, and C is called *connector*, is a pushout of pair $j : B \leftarrow C \rightarrow C \times W : \langle 1_C, e \rangle$ provided that it exists (implying that product $C \times W$ exists as well) and is preserved by functor *str*. □

This definition captures intuitive properties of weaving. For example, if B consists solely of join points (i.e. j is an isomorphism), then weaving produces a product $B \times W$. Labeled scenarios (i.e. objects of a category *AO* constructed from constituents of scenario modeling school *SM*) are friendly to weaving. In particular, weaving exists iff the connector "tolerates" concurrency in a sense that it doesn't impose specific order of executing different aspects of the advice bound to the same join point. Formally, for every $x, y \in mod(C)$ conditions $mod(j)(x) = mod(j)(y)$ and $x \le y$ shall imply that $asp(W)(v) = asp(W)(x)$ for every such $v \in mod(W)$ that $mod(e)(x) \le v \le mod(e)(y)$. This holds for weavers with implicit connectors, such as AspectJ.

The construction of weaving suggests how to extract individual aspects from multiaspect program. The category-theoretic construction of a pullback (dual to a pushout) is employed there. Recall that a pullback is a limit (dual to a colimit) of a diagram that has a form of two arrows with the same destination. A pullback is used to generalize set-theoretic notion of a preimage of a subset: given a diagram $s : S \to A \leftarrow B : f$, where s identifies a subobject S in A, and its pullback $p : S \leftarrow P \rightarrow B : q$, morphism q identifies a "preimage" $f^{-1}(S)$. Similarly, a subaspect of an *AO*-object A is essentially a preimage of its aspect structure along a traceable refinement represented by $asp(A)$.

Sound notion of a subaspect allows formal evaluation of modularizing crosscutting concerns, viz. separating them into modular design units. The first step towards modularization consists in *explicating* aspect structure of an *AO*-object as a traceable refinement. Although it may be impossible or ambiguous due to tangling, each nonempty AO-school contains models that allow naturally explicating their aspect structures as well as integration actions.

Definition 7. An *explication* (of aspect structure) of an *AO*-description S is an *r-DESC*-morphism $s : X \to mod(S)$ that is dual to a *sig*-trace and satisfies equality $sig(s^{op}) = asp(S)$. An explication s is called *universal* provided that for every *AO*-morphism $f : S \to R$ and every explication r of aspect structure of R there exists a *c-DESC*-morphism q, called *explication* of f along r, such that $q \circ s^{op} = r^{op} \circ mod(f)$. An (aspectual) *core* of an AO-school is full subcategory of *AO* that consists of all descriptions that have a universal explication. □

Obviously a universal explication is unique up to an isomorphism. Moreover, observe that the explication equality resembles the definition of a natural transformation. This is not a mere coincidence: explicating an *AO*-morphism is actually a functor, and universal explications comprise natural transformation of functor *mod* (reduced on the core) to it. An example of a core *AO*-object is a pair $\langle A, 1_{sig(A)} \rangle$ obtained from a *c-DESC*-object A by functor *mod**.

Once an aspect structure of an *AO*-description is explicated as a refinement, individual aspects need to be extracted from it for subsequent modular development. Partitioning complex models to extractable aspects is known as *separation of concerns*. A key to separation is obtaining *AO*-morphisms with pullbacks as explications.

Definition 8. An *AO*-morphism $m : A \to S$ is called a *subaspect* of a core *AO*-description S if it satisfies the following conditions:

- A is a core aspect;
- explication m' of m is right inverse to a trace;
- explication equality $m' \circ a^{op} = s^{op} \circ mod(m)$, where a and s are universal explications of A and S, respectively, determines a *c-DESC*-pullback. □

In an architecture school over **Set**, explication of aspect structure of an *AO*-description S consists in equipping set $str(S)$ with enough "modular" structure to turn map $asp(S)$ into actual trace directed from $mod(S)$. If such equipping is possible, then a candidate subaspect in S can be identified, like in **Set**, by pulling back a (weak) element (i.e. a morphism whose domain rests upon a singleton set) along this trace. An underlying set of the pullback object is precisely an equivalence class of aspect structure equivalence relation. In order for it to form a genuine subaspect, both it and the codomain of the identifying element should be produced from the element's domain by traceable refinements.

Every core labeled scenario can be partitioned to subaspects. However, the core is rather "small": for example, two linearly ordered aspects executed in interleaving mode cannot be separated from each other. Weaving cannot directly produce interleaving as well. This fact illustrates difficulties encountered at developing even simple client-server distributed systems. Yet every scenario can be labeled by linearly ordered extractable aspects. Their number can be either maximized by applying the functor *mod**, or minimized by identifying so-called sequential subsystems [8]. This fact justifies developing aspect-oriented extensions to traditional programming languages that allow creating only sequential programs.

As an example application of aspect-oriented scenario modeling, consider a distributed measurement system (DMS). As shown in [9], its main execution scenario consists in reiterating the following linearly ordered sequence of data processing (functional) concerns:

$$measure \rightarrow store \rightarrow validate \rightarrow compute \rightarrow display.$$

During system development, each of them is refined to a complex aspect, yet they remain separable. However, infrastructure aspects, such as metadata model, monitoring, and security, are woven to each of them, undermining separation of concerns. So in order to execute different data processing functions on different computers, the infrastructure has to be somehow replicated among them. It is this replication that makes a DMS considerably more challenging to develop than an isolated measurement device.

A glance on results of this section reveals that major contribution of AOSD into software design (in its category-theoretic treatment) consists in employing various kinds of limits (constructions dual to colimits), including a terminal object, products, and pullbacks. Contrast this with traditional modular design that, as presented in Section 2, is based solely on colimits. The root reason of limits to appear is of course the duality between integration actions and traceable refinements, as imposed by Definition 1.

5. CONCLUSION

Our work belongs to the mainstream of applications of category theory to computer science. Their success is due to ability of category-theoretic notions to formally express basic mental patterns of systems analysis, which is the crucial software design activity. In particular, fundamental results were achieved in the area of "categorizing" modular design. However, to the best of our knowledge there are no comparably powerful frameworks suitable to construct and analyze aspect-oriented development technologies. Existing AOSD methods are represented in terms of concrete formal devices difficult to apply beyond specific software development paradigms. Formalisms employed to express aspect-oriented concepts include process algebras [1], model checking [6], architecture description languages [12], graph transformations [17], and so on. In contrast to them, our approach aims at producing aspect-oriented methods suitable for any particular designers' needs by formal transformation of a given modular architecture.

So far presented metamodel is too abstract to be directly applied in software development. Its instances pertaining to major existing design technologies need to be developed and generously illustrated with examples. Such kind of development is a promising area of further research. Much work also has to be done in discovering capabilities and limitations of AO-schools, creating abstract yet powerful aspect weaving and separating techniques.

6. REFERENCES

[1] Andrews J.H. Process-Algebraic Foundations of Aspect-Oriented Programming. Lecture Notes in Computer Science, Vol. 2192, 2001, 187–209.

[2] Aspect-Oriented Software Development. Addison Wesley, 2004.

[3] Colyer A., Clement A., Harley G., Webster M. Eclipse AspectJ. Addison-Wesley, 2004.

[4] Fiadeiro J.L., Lopes A., Wermelinger M. A Mathematical Semantics for Architectural Connectors. Lecture Notes in Computer Science, Vol. 2793, 2003, 190–234.

[5] Glabbeek R.J. van, Goltz U. Refinement of Actions and Equivalence Notions for Concurrent Systems. Acta Informatica, Vol. 37, Issue 4–5, 2000, 229–327.

[6] Katz E., Katz S. Verifying Scenario-Based Aspect Specifications. Lecture Notes in Computer Science, Vol. 3582, 2005, 432–447.

[7] Kiczales G. et al. Aspect-Oriented Programming. Lecture Notes in Computer Science, Vol. 1241, 1997, 220–242.

[8] Kovalyov S.P. Architecture of Time of Distributed Information Systems. J. Computational Technologies, Vol. 7, No. 6, 2002, 38–53. [In Russian]

[9] Kovalyov S.P. Domain Engineering of Distributed Measurement Systems. Optoelectronics, Instrumentation and Data Processing, 44(2), 2008, 125–130.

[10] Lopes A., Fiadeiro J.L. Revisiting the Categorical Approach to Systems. Lecture Notes in Computer Science, Vol. 2422, 2002, 426–440.

[11] Mac Lane S. Categories for Working Mathematician. 2nd Ed. Springer, 2008.

[12] Pinto M., Fuentes L., Troya J.M. DAOP-ADL: An Architecture Description Language for Dynamic Component and Aspect-Based Development. Lecture Notes in Computer Science, Vol. 2830, 2003, 118–137.

[13] Pratt V.R. Modeling Concurrency with Partial Orders. Intl. J. Parallel Programming, 15(1), 1986, 33–71.

[14] Sassone V., Nielsen M., Winskell G. Deterministic Behavioural Models for Concurrency. Lecture Notes in Computer Science, Vol. 711, 1993, 682–692.

[15] Steimann F. The Paradoxical Success of Aspect-Oriented Programming. In Proceedings of OOPSLA'06. Portland, 2006, 481–497.

[16] Vanhooff B., Baelen S. van, Joosen W., Berbers Y. Traceability as Input for Model Transformations. In Proceedings of the 3rd ECMDA-TW. Haifa, Israel, 2007, 37–46.

[17] Whittle J., Jayaraman P. MATA: A Tool for Aspect-Oriented Modeling Based on Graph Transformation. Lecture Notes in Computer Science, Vol. 5002, 2008, 16–27.