Pluggable AOP — Designing Aspect Mechanisms for Third-party Composition*

Sergei Kojarski†
Northeastern University
Boston, Massachusetts 02115-5000, USA
kojarski@cs.virginia.edu

David H. Lorenz‡
University of Virginia
Charlottesville, Virginia 22904-4740, USA
lorenz@cs.virginia.edu

ABSTRACT
Studies of Aspect-Oriented Programming (AOP) usually focus on a language in which a specific aspect extension is integrated with a base language. Languages specified in this manner have a fixed, non-extensible AOP functionality. This paper argues the need for AOP to support the integration and use of multiple domain-specific aspect extensions together. We study the more general case of integrating a base language with a set of third-party aspect extensions for that language. We present a general mixin-based semantic framework for implementing dynamic aspect extensions in such a way that multiple, independently developed aspect mechanisms can be subject to third-party composition and work collaboratively. Principles governing the design of a collaborative aspect mechanism are aspectual effect exposure and implementation hiding.

Categories and Subject Descriptors
D.2.10 [**Software Engineering**]: Design—*Design concepts*; D.1.5 [**Programming Techniques**]: Aspect-oriented Programming; D.3.1 [**Programming Languages**]: Formal Definitions and Theory; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages; D.2.12 [**Software Engineering**]: Interoperability

General Terms
Design, Languages, Theory

Keywords
AOP, AOSD, domain-specific aspect language, aspect extension, aspect mechanism, aspectual effect, granularity, software components, CBSE, third-party composition, collaboration, reuse, mixin, semantics, AspectJ, AspectWerkz, COOL.

*This research is supported in part by NSF's Science of Design program under Grant Number CCF-0438971.

†Sergei Kojarski is a PhD candidate at Northeastern University and a visiting graduate student at University of Virginia.

‡Research done in part while at Northeastern University.

OOPSLA'05, October 16–20, 2005, San Diego, California, USA.

1. INTRODUCTION
A current trend in Aspect-Oriented Programming (AOP [27]) is the usage of general-purpose AOP languages (AOPLs). However, a general-purpose AOPL lacks the expressiveness to tackle all cases of crosscutting. A solution to unanticipated crosscutting concerns is to create and combine different domain-specific aspect extensions to form new AOP functionality. As of yet, there is no methodology to facilitate this process [43].

Studies of AOP typically consider the semantics for an AOPL that integrates a certain aspect extension, Ext_1, with a base language, Base. For example, Ext_1 might be (a simplified version of) AspectJ [26] and Base (a simplified version of) Java [1]. The semantics for the integration $\mathsf{Base} \times \mathsf{Ext}_1$ is achieved by amending the semantics for the base language. Given a pair of programs $\langle base, aspect_1 \rangle \in \mathsf{Base} \times \mathsf{Ext}_1$, the amended semantics explains the meaning of $base$ in the presence of $aspect_1$.

Unfortunately, the semantics for the aspect extension and that for the base language become tangled in the process of integration. Consequently, it is difficult to reuse or combine aspect extensions. For each newly introduced aspect extension, say Ext_2, the semantics for $\mathsf{Base} \times \mathsf{Ext}_2$ needs to be reworked. Moreover, given the semantics for $\mathsf{Base} \times \mathsf{Ext}_1$ and the semantics for $\mathsf{Base} \times \mathsf{Ext}_2$, the semantics for $\mathsf{Base} \times \mathsf{Ext}_1 \times \mathsf{Ext}_2$ is undefined even though Ext_1 and Ext_2 are both aspect extensions to the same base language.

In this paper we resolve this difficulty by considering a more general open question:

THE ASPECT EXTENSION COMPOSITION QUESTION: *Given a base language, Base, and a set $\{\mathsf{Ext}_1, \ldots, \mathsf{Ext}_n\}$ of independent aspect extensions to Base, what is the meaning of a program $base \in \mathsf{Base}$ in the base language in the presence of n aspect programs $\langle aspect_1, \ldots, aspect_n \rangle \in \mathsf{Ext}_1 \times \cdots \times \mathsf{Ext}_n$ written in the n different aspect extensions?*

Ability to compose distinct aspect extensions offers first and mostly great practical benefits. In Section 2 we discuss these benefits and illustrate that, unfortunately, prevalent implementations are not composable. Addressing the general composition question also provides in the special case where $n = 1$ a better encapsulation of the semantics for a single aspect extension.

1.1 Combining Two Aspect Extensions
Answering the aspect extension composition question is difficult even for $n = 2$. Let MyBase be a procedural language, and con-

sider two independent, third-party aspect extensions to MyBase. The first, HisExt_1, capable of intercepting procedure calls and similar in flavor to AspectJ. The other, HerExt_2, an aspect extension to MyBase capable of intercepting calls to the primitive division operator for catching a division by zero before it even happens (as opposed to catching a division by zero exception after it occurs), a capability that AspectJ lacks.[1] Both call interception (e.g., [28]) and checking if a divisor is zero (e.g., [3, 29, 17]) are benchmarks often used in connection with aspects.

W.l.o.g., assume HisExt_1 is created before HerExt_2 is even conceived. If HisExt_1 is to eventually work collaboratively with another aspect extension, e.g., HerExt_2, the implementation of HisExt_1 must take special care to expose its AOP effect, *and only* its effect, in terms of MyBase. This is because an $aspect_2$ program written in HerExt_2 would need to intercept divisions by zero not only in the base program $base$ but also in advice introduced by an $aspect_1$ program written in HisExt_1.

Failing to reify a division by zero in $aspect_1$ might cause a false-negative effect in HerExt_2. Meanwhile, $aspect_2$ must not intercept divisions by zero, if any, in the implementation mechanism of either HisExt_1 or HerExt_2. Reifying a division by zero in the implementation mechanism might cause a false-positive effect in HerExt_2.

Similarly, $aspect_1$ must intercept not only procedure calls in $base$ but also any matching procedure call introduced by $aspect_2$. $aspect_1$ must not, however, intercept internal procedure calls that are a part of the implementation mechanism of either HisExt_1 or HerExt_2.

Note that generally aspect extensions present incompatible levels of AOP granularity [31]. In our example, $aspect_1$ is not expressible in HerExt_2, and $aspect_2$ is not expressible in HisExt_1. Therefore the problem of integrating the two cannot be reduced to translating $aspect_1$ to HerExt_2 or translating $aspect_2$ to HisExt_1 and using just one aspect extension. This distinguishes our objective from the purpose of frameworks (like XAspects [39]) that rely on the use of a general-purpose AOPL (like AspectJ).

In the sequel, a *base mechanism* denotes an implementation of a base language semantics, an *aspect mechanism* denotes an implementation of an aspect extension semantics, and a *multi mechanism* denotes an implementation of a multi-extension AOPL.

1.2 Objective and Contribution

We describe a general method for implementing the base mechanism and the aspect mechanisms in such a way that multiple, independent aspect mechanisms can be subject to third-party composition and work collaboratively. By third-party composition of aspect mechanisms we mean a semantical framework in which distinct aspect mechanisms can be assembled with the base mechanism into a meaningful multi mechanism without modifying the individual mechanisms. The mechanisms are said to be collaborative units of composition if the semantics of the composed multi mechanism can be derived from the semantics of the mechanisms that comprise it.

More precisely, let $\mathcal{B}$ denote the base mechanism for Base. Let $\mathcal{M}_1, \ldots, \mathcal{M}_n$ denote the aspect mechanisms for $\mathsf{Ext}_1, \ldots, \mathsf{Ext}_n$, respectively. The ***aspect mechanism composition problem*** is to enable the third-party composition of $\mathcal{M}_1, \ldots, \mathcal{M}_n$ with $\mathcal{B}$ into a multi mechanism $\mathcal{A}$, in a manner similar to the assembly of software components:[2]

- **Units of independent production.** The aspect mechanisms $\mathcal{M}_1, \ldots, \mathcal{M}_n$ are independently defined. The base mechanism $\mathcal{B}$ is defined independently from $\mathcal{M}_1, \ldots, \mathcal{M}_n$. To enable the composition, $\mathcal{M}_1, \ldots, \mathcal{M}_n$ rely only on $\mathcal{B}$ and have an explicit context dependency only on $\mathcal{A}$.
- **Units of composition.** The mechanisms are subject to third-party composition. The multi mechanism $\mathcal{A}$ for the combined AOP language is constructed (denoted by a $\boxplus$ combinator) by composing the base mechanism with the aspect mechanisms without altering them: $\mathcal{A} = \boxplus\langle\mathcal{B}, \mathcal{M}_1, \ldots, \mathcal{M}_n\rangle$
- **Units of collaboration.** The semantics for the composed multi mechanism $\mathcal{A}$ is the "sum" of the semantics provided by all the mechanisms.

Independence enables third-party development of individual aspect mechanisms; composability enables third-party composition of aspect mechanisms; and collaboration enables the desired behavior in the constructed AOP language.

Specifically, our approach enables third-party composition of dynamic aspect mechanism. We illustrate our solution for expression evaluation semantics. We model each aspect mechanism as a transformation function that revises the evaluation semantics for expressions.

1.3 Outline

In the rest of this paper, we demonstrate our solution to the aspect mechanism composition problem concretely through the implementation of interpreters. The next section motivates the need for composing multiple aspect extensions and demonstrates the lack of integration support in current aspect mechanisms. Section 3 presents a concrete instance of the problem: a base language MyBase with two aspect extensions, HisExt_1 and HerExt_2. We present their syntax and analyze a runnable programming example implemented in our framework. In Section 4 we present our approach for the general case of integrating n aspect mechanisms. In Section 5 we revisit the example shown in Section 3 and formally demonstrate our approach by constructing the semantics for MyBase, HisExt_1, and HerExt_2.

2. MOTIVATION

There is a growing need for the simultaneous use of multiple domain-specific aspect extensions. The need steams mainly from the favorable trade-offs that a domain-specific aspect extension can offer over a general-purpose AOPL:

- *Abstraction.* A general-purpose AOPL offers low-level abstractions for covering a wide range of crosscutting concerns. Domain-specific aspect extensions, in contrast, can offer abstractions more appropriate for the crosscutting cases in the domain at hand, letting the programmer concentrate on the problem, rather than on low-level details.

[1] AspectJ can neither advise primitives nor arguments.

[2] A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to third-party composition [41].

- *Granularity.* The granularity of an aspect extension dictates all possible concern effect points within an application. Combining domain-specific aspect extensions allows to overcome the fixed granularity limitation of general-purpose AOPLs [31].

- *Expressiveness vs. Complexity.* The granularity of a general-purpose AOPL exposes a non-linear relationship between the language expressiveness and complexity. An increase in the language granularity significantly increases the language complexity while achieving a relatively small increase in expressiveness. Domain-specific aspect extensions, in contrast, can offer independent diverse ontologies [49].

The need also arises from the sheer abundance of available aspect extensions (and their evolving aspect libraries). For the Java programming language alone there are numerous aspect extensions that are being used in a variety of commercial and research projects. These include: AspectJ (ajc [10] and abc [2]), AspectWerkz [4], COOL [30], JBoss-AOP [25], JAsCo [44], Object Teams [20], ComposeJ [51], to name just a few.[3] Ability to use these aspect extensions together will allow to reuse existing (and future) aspect libraries written for the different aspect extensions.

Unfortunately, little support is provided for the integration of distinct aspect mechanisms. Each aspect mechanism creates its own unique program representation which often excludes foreign aspects. Consequently, interaction between multiple aspect mechanisms operating on a single program can produce unexpected or incoherent results.

2.1 Example

Consider a bounded buffer example implemented in Java (Listing 1). Suppose you have three aspect extensions at your disposal:

- COOL [30]—a domain-specific aspect extension for expressing coordination of threads;

- AspectWerkz [4]—a general-purpose lightweight AOP framework for Java;

- AspectJ—a general-purpose aspect extension;

and two concerns to address, namely, a synchronization concern and a tracing concern (e.g., logging or auditing).

2.1.1 COOL versus AspectJ

The synchronization concern can be expressed as a coordinator aspect in COOL (e.g., Listing 2) or alternatively as an aspect in AspectJ (e.g., Listing 3).

The COOL aspect (Listing 2) provides an elegant declarative description of the desired synchronization. The **mutex** exclusion set `{add, remove}` specifies that `add` may not be executed by a thread while `remove` is being executed by a different thread, and vice versa. In addition, the **selfex** exclusion set prohibits different threads from simultaneously executing either `add` or `remove`.[4]

[3]For a complete list of commercial and research aspect extensions see `http://www.aosd.net/technology/`

[4]However, the same thread is not prohibited from entering both `add` and `remove`.

Listing 1: A non-synchronized bounded buffer

```
1  public class BoundedBuffer {
2
3    private Object[] buffer;
4    private int usedSlots = 0;
5    private int writePos = 0;
6    private int readPos = 0;
7    private static BoundedBuffer singltn = null;
8
9    public static BoundedBuffer getInstance() {
10     return singltn;
11   }
12
13   public BoundedBuffer (int capacity) {
14     this.buffer = new Object[capacity];
15     singltn = this;
16   }
17
18   public Object remove() {
19     if (usedSlots == 0) {return null;}
20     Object result = buffer[readPos];
21     buffer[readPos] = null;
22     usedSlots--; readPos++;
23     if (readPos==buffer.length) readPos=0;
24     return result;
25   }
26
27   public void add(Object obj) throws Exception {
28     if (usedSlots==buffer.length)
29       throw new Exception("buffer is full");
30     buffer[writePos] = obj;
31     usedSlots++;
32     writePos++;
33     if (writePos==buffer.length) writePos=0;
34   }
35 }
```

The COOL code is expressive, concise, readable, and easy to understand. It provides the right abstractions. Studies [34, 47, 33, 48] have shown that "participants could look at COOL code and understand its effect without having to analyze vast parts of the rest of the code," and that "COOL as a synchronization aspect language eased the debugging of multi-threaded programs, compared to the ability to debug the same program written in Java" [46].

While it is possible to express the same concern in AspectJ, the code will be much longer. In comparison to the COOL code, the AspectJ implementation (Listing 3) requires 10 times more lines of code. It is also harder to explain and understand. The aspect implements a monitor using two condition variables `remove_thread` and `add_thread`. Using two pieces of **around execution** advice, the aspect obtains locks (`remove_thread` and `add_thread`) for the duration of executing **proceed** (execution of `remove` and `add`, respectively). This guaranties that no more than one thread operates on the buffer at a time. If `remove_thread` or `add_thread` are locked by some other thread, the advice waits. When the thread has a lock, it runs **proceed** and afterwards releases the lock by signaling `notifyAll()`, which in turn wakes up other waiting threads.

2.1.2 AspectWerkz + AspectJ

Semantically, the underlying mechanisms of AspectWerkz and AspectJ are essentially equivalent. Yet, their syntactical differences present programmers with a useful choice of alternatives. Recently it was announced that AspectWerkz has joined the AspectJ project

Listing 2: Synchronization aspect in COOL

```
coordinator BoundedBuffer {
  selfex {add, remove},
  mutex {add, remove};
}
```

Listing 3: Synchronization aspect in AspectJ

```
public aspect BufferSyncAspect {
  private Object remove_thread=null;
  private Object add_thread=null;

  Object around():
    execution(Object BoundedBuffer.remove()) {
    Object this_thread = Thread.currentThread();
    synchronized(this) {
      while ((remove_thread!=null &&
           remove_thread!=this_thread) ||
         (add_thread!=null &&
          add_thread!=this_thread))
        try {wait();
        } catch (InterruptedException e) {}
      remove_thread = this_thread;
    }
    Object result = proceed();
    synchronized(this) {
      remove_thread = null;
      notifyAll();
    }
    return result;
  }

  void around() throws Exception:
    execution(void BoundedBuffer.add(Object)) {
    Object this_thread = Thread.currentThread();
    synchronized(this) {
      while ((remove_thread!=null &&
           remove_thread!=this_thread) ||
         (add_thread!=null &&
          add_thread!=this_thread))
        try {wait();
        } catch (InterruptedException e) {}
      add_thread = this_thread;
    }
    try{proceed();}
    finally {
      synchronized(this) {
        add_thread = null;
        notifyAll();
      }
    }
  }
}
```

to bring the key features of AspectWerkz to the AspectJ 5 platform [5]. This merger will allow aspects like those in Listing 4 and Listing 5 to run side by side.

Listing 4 is a simple logging aspect in AspectWerkz. The annotation *@Aspect(”perJVM”)* specifies that the `AWLogger` class is actually a singleton aspect. The annotation *@Before call(* *.*(..)) && !cflow(within(AWLogger))* specifies that the `log` method is to be called for every method call not in the dynamic control flow of methods in `AWLogger`.

Listing 5 is an auditing aspect in AspectJ. The `toLog()` pointcut specifies that every method call should be recorded. The **`before`**, **`after()returning`**, and **`after()throwing`** advice add log messages to the buffer.

Listing 4: Logging aspect in AspectWerkz

```
/** @Aspect(”perJVM”) */
public class AWLogger {
  /**@Before call(* *.*(..))&&!cflow(within(AWLogger))*/
  public void log(JoinPoint jp) {
    System.out.println("AW:"+jp.getSignature());
  }
}
```

Listing 5: Auditing aspect in AspectJ

```
public aspect AJAuditor {

  pointcut toLog():
    call(* *.*(..)) && !cflow(within(AJAuditor));

  before(): toLog() {
    log("ENTER",thisJoinPoint);
  }
  after() returning: toLog() {
    log("EXIT",thisJoinPoint);
  }
  after() throwing: toLog() {
    log("THROW",thisJoinPoint);
  }

  protected void log(String aType,JoinPoint jp) {
    BoundedBuffer buf=BoundedBuffer.getInstance();
    if (buf==null) return;
    try{buf.add(jp);} catch (Exception e) {
      System.out.println(e.getMessage());
    }
  }
}
```

Arguably, if AspectWerkz and AspectJ were designed to be composable third-party aspect mechanisms, building AspectJ 5 would have been much easier. Moreover, third-party composition of aspect mechanisms would have made other domain-specific combinations possible, like combining COOL with AspectWerkz and AspectJ.

2.2 Lack of Integration Support

Unfortunately, current aspect mechanisms fail to compose correctly. We demonstrate this failure on the bounded buffer example for two commonly used approaches:

- *Translation.* Aspect programs in different aspect extensions can be translated to a common target aspect extension.
- *Instrumentation.* Aspect mechanisms can be implemented by means of program instrumentation. Such multiple independent aspect mechanisms can be trivially composed by passing the output of one aspect mechanism as the input to another aspect mechanism.

2.2.1 No Behavior-Preserving Translation

The translation approach requires the expressiveness of the target aspect extension to support arbitrary granularity. Even when granularity does not pose a problem, a translation from one aspect language to another will not generally preserve the behavior of the

source aspect program in the presence of other aspects. Consider the synchronization concern implementation in COOL (Listing 2). Translating it to AspectJ (Listing 3) results in an aspect that seems to be a correct substitution for the COOL coordination aspect, but in the presence of the auditing aspect (Listing 5) is actually not.

A property of the COOL synchronization concern is transparency with respect to the AspectJ auditing concerns. There should not be any interference between the two. The COOL aspect does not contain any join points that should be visible to the AspectJ mechanism. This property is not preserved in the translation. Calls to `wait` (Listing 3, lines 13 and 33) and `notifyAll` (Listing 3, lines 20 and 41), which do not exist in the COOL code, will nonetheless be unexpectedly reflected by the auditor.[5]

Worse yet, the unexpected join points in the target program may break existing invariants, resulting in our case in a deadlock. An implicit invariant of the COOL aspect is that if both `add` and `remove` are not currently executed by some other thread, then the thread can enter and execute them. The AspectJ synchronization aspect, however, violates this invariant. Assume that two threads concurrently access the buffer. The first thread acquires the lock, while the second invokes `wait` on the `BufferSyncAspect` object. However, before `wait` is invoked, the `AJAuditor` aspect calls `BoundedBuffer.add` (Listing 5, line 19). The latter call causes the second thread to enter the guarded code *again* and trigger a *second* call to `wait`.[6] Since the second `wait` call is in the **`cflow`** of the auditor, it is not advised, and the thread finally suspends. When the first thread releases the lock, the second thread wakes up after the second `wait`. It acquires the lock, completes the advice execution, releases the lock, and proceeds to the *first* `wait` invocation. At this point, the buffer is not locked; the second thread waits on the `BufferSyncAspect` object monitor; and if no other thread ever accesses the buffer, the second thread waits for ever—deadlock!

2.2.2 No Correct Sequential Instrumentation

One would expect the two aspects written in AspectWerkz (Listing 4) and AspectJ (Listing 5) to interact as if they were two aspects written in a single aspect extension (e.g., the future AspectJ 5 platform). On the one hand, the AspectJ auditor should log all method calls within the `AWLogger` aspect. On the other hand, the AspectWerkz logger should log all method calls within `AJAuditor`. (And both should log all method calls in the base program as well.)

However, applying the AspectJ and AspectWerkz instrumentation mechanisms sequentially, in any order, produces an unexpected result. The mechanism that is run first may not be able to interpret the second extension's aspect program. Specifically, the AspectWerkz mechanism does not understand AspectJ's syntax. It can be applied to the bounded buffer code but not to the `AJAuditor` aspect. Thus, when AspectWerkz is run first, some expected log messages will be missing.

The mechanism that is run last logs method calls that are not supposed to be logged. For example, when AspectWerkz is run second, the following unexpected log message is generated by the `AWLogger` aspect:

```
AW:public void AJAuditor.ajc$afterReturning$-
AJAuditor$2$ba1fbd8a(org.aspectj.lang.JoinPoint)
```

[5]Note that calls to `wait` and `notifyAll` cannot be avoided.

[6]Assuming that the first thread still owns the lock.

3. PROBLEM INSTANCE

We now return to MyBase, HisExt_1, and HerExt_2 in order to analyze the problem and illustrate our approach concretely. After a brief introduction to the syntax, we informally explain MyBase, HisExt_1, and HerExt_2 through a programming example. The code fragments are actual running code in our implementation, and their semantics is formally presented in Section 5.

3.1 Syntax

3.1.1 MyBase *Syntax*

The syntax of MyBase is given in Figure 1. MyBase is a procedural language. Procedures are mutually-recursive with call-by-value semantics. The set of procedures is immutable at run-time. Expressed values are either booleans or numbers (but not procedures). The execution of a program starts by evaluating the body of a procedure named **`main`**.

3.1.2 HisExt_1 *Syntax*

The syntax for HisExt_1 is given in Figure 2. HisExt_1 is a simple AspectJ-like aspect extension to MyBase. HisExt_1 allows one to impose advice around procedure calls and procedure executions. Advice code is declared in a manner similar to procedures. Like in AspectJ, the set of advice is immutable at run-time. Each advice has two parts: a pointcut designator and an advice body expression. Atomic pointcuts are `pcall-pcd`, `pexecution-pcd`, `cflow-pcd`, and `args-pcd`. The `and-pcd` and `or-pcd` allows one to combine several pointcuts under conjunction and disjunction, respectively. Unlike AspectJ, `around` is the only advice kind in HisExt_1. There is no support for patterns in pointcut designators.

HisExt_1 introduces a new `proceed-exp` expression. The notation:

$$\texttt{Exps} ::= \texttt{...} \mid \texttt{proceed-exp}$$

redefines `Exps` within the aspect extension syntax only, without propagating the change to the syntax of MyBase expressions. In particular, `proceed-exp` expressions are valid only within a HisExt_1 advice-body expression.

3.1.3 HerExt_2 *Syntax*

HerExt_2 allows one to declare a set of exception handlers in MyBase for catching and handling division by zero before an exception occurs. Advice code in HerExt_2 specifies an exception handler expression. A guard clause allows one to specify a dynamic scope for the handler. HerExt_2 introduces a new expression, namely `raise-exp`, which is allowed within a handler. It passes the exception handling to the next handler (in a manner, similar to `proceed-exp` of HisExt_1). The syntax of the language is given in Figure 3.

The semantics for HerExt_2 is straightforward. Whenever the second argument to the division primitive evaluates to zero, the advice handler (if one exits) is invoked. The handler is evaluated and the result value substitutes the offending zero in the second argument to the division primitive, and the program execution resumes.

Listing 8 shows an aspect we can write in HerExt_2. This aspect resumes the execution with the value of `Precision(1)` whenever the second argument of a division primitive evaluates to 0 within the control flow of the SQRT procedure.

`Program`	::=	`Declaration`	Program
`Declaration`	::=	"**`program`**" "{" `Procedure`* "}"	Declaration
`Procedure`	::=	"**`procedure`**" `PName` "**(**" `Id`* "**)**" `Exps`	Procedure
`Exps`	::=	`lit-exp` | `true-exp` | `false-exp` | `var-exp` | `app-exp` | `begin-exp` | `if-exp` | `assign-exp` | `let-exp` | `primapp-exp`	Expressions
`lit-exp`	::=	`Number`	Numbers
`true-exp`	::=	"**`true`**"	True
`false-exp`	::=	"**`false`**"	False
`var-exp`	::=	`Id`	Id meaning
`app-exp`	::=	"**`call`**" `PName` "**(**" `Exps`* "**)**"	Procedure call
`begin-exp`	::=	"{" `Exps` ("**;**" `Exps`)* "}"	Block
`if-exp`	::=	"**`if`**" `Exps` "**`then`**" `Exps` "**`else`**" `Exps`	Conditional
`assign-exp`	::=	"`set`" `Id` "=" `Exps`	Assignment
`let-exp`	::=	"**`let`**" (`Id` "=" `Exps`)* "**`in`**" `Exps`	Let
`primapp-exp`	::=	`Prim` "**(**" `Exps`* "**)**"	Primitive application
`Prim`	::=	"**+**" | "**-**" | "*****" | "/" | "**lt?**" | "**eq?**"	Primitives
`Id`			Identifier
`PName`			Procedure name
`Number`			Numbers

Figure 1: MyBase syntax

`AOP1-Program`	::=	`AOP1-Declaration`	HisExt$_1$ program
`AOP1-Declaration`	::=	"**`aop1`**" "{" `Advice`* "}"	HisExt$_1$ declaration
`Advice`	::=	"**`around`**" "**:**" `Pointcut` `Exps`	Advice
`Pointcut`	::=	`call-pcd` | `exec-pcd` | `cflow-pcd` | `args-pcd` | `and-pcd` | `or-pcd`	Pointcut designators
`call-pcd`	::=	"**`pcall`**" "**(**" `PName` "**)**"	Procedure call pcd
`exec-pcd`	::=	"**`pexecution`**" "**(**" `PName` "**)**"	Procedure execution pcd
`cflow-pcd`	::=	"**`cflow`**" "**(**" `PName` "**)**"	Control flow pcd
`args-pcd`	::=	"**`args`**" "**(**" `Id`* "**)**"	Argument pcd
`and-pcd`	::=	"**`and`**" "**(**" `Pointcut`* "**)**"	Conjunction pcd
`or-pcd`	::=	"**`or`**" "**(**" `Pointcut`* "**)**"	Disjunction pcd
`Exps`	::=	`...` | `proceed-exp`	Advice expressions
`proceed-exp`	::=	"**`proceed`**"	Proceed exp

Figure 2: HisExt$_1$ syntax

`AOP2-Program`	::=	`AOP2-Declaration`	HerExt$_2$ program
`AOP2-Declaration`	::=	"**`aop2`**" "{" `Handler`* "}"	HerExt$_2$ declaration
`Handler`	::=	"**`guard_cflow`**" `PName` "**`resume_with`**" `Exps`	Handlers
`Exps`	::=	`...` | `raise-exp`	Handler expressions
`raise-exp`	::=	"**`raise`**"	Raise expressions

Figure 3: HerExt$_2$ syntax

Listing 6: A naive program in MyBase for computing $\sqrt{x}$

```
101 program {
102  procedure SQRT(radicand) {
103   call SqrtIter(0,radicand,call Precision(radicand
        ))
104  }
105  procedure SqrtIter(approximation,radicand,
        precision) {
106   let
107    bid = call Improve(approximation,radicand,
          precision)
108   in
109    if call IsPreciseEnough?(bid,radicand)
110    then bid
111    else call SqrtIter(bid,radicand,precision)
112  }
113  procedure Improve(approximation,radicand,
        precision) {
114   +(approximation,precision)
115  }
116  procedure Precision(x) {1}
117  procedure IsPreciseEnough?(root,square) {
118   lt?(square,call Square(root))
119  }
120  procedure Square(x) {*(x,x)}
121  procedure Abs(x) {if lt?(x,0) then -(0,x) else x}
122  procedure main() {call SQRT(5)}
123 }
```

Listing 7: Advice in HisExt$_1$ for using Newton's method

```
201 aop1 {
202  around: and(pexecution(Improve) args(an,x,epsilon)) {
203   /(+(an,/(x,an)),2)
204  }
205  around: and(pexecution(IsPreciseEnough?) args(root,x)
        ) {
206   lt? (call Abs(-(x,call Square(root))),call
        Precision(x))
207  }
208  around : pcall(Precision) {
209   /(proceed,1000)
210  }
211 }
```

Listing 8: Advice in HerExt$_2$ for preventing an exception

```
301 aop2 { guard_cflow SQRT resume_with call Precision(1) }
```

3.2 A Programming Example

The semantics for the base procedural language MyBase and the aspect extensions HisExt$_1$ and HerExt$_2$ are implemented as interpreters [18]. The example presented here is a simple executable arithmetic program in MyBase for computing the square root of a given number. While simple, the example is representative in terms of illustrating the complexity of achieving collaboration among aspect extensions, and its semantics serves as a proof of concept.

The procedure SQRT in Listing 6 implements in MyBase a simple approximation algorithm using a sequence generated by a recurrence relation:

$$a_0{=}approximation\ ;\ \textbf{repeat}\ a_n{=}f(a_{n-1})\ \textbf{until}\ precise$$

By default, it sets $a_0 = 0$, and calls `SqrtIter` to generate the recurrence sequence:

$$a_n = a_{n-1} + \epsilon$$

until $(a_n)^2 > x$. The procedure `Improve` generates the next element in the sequence; `IsPreciseEnough?` checks the termination condition; and the value $\epsilon = \epsilon(x)$ is computed as a function of x by the procedure `Precision`.

The resulted computation of $\sqrt{x}$ is inaccurate and extremely inefficient. However, it serves our purpose well. We will non-intrusively improve its efficiency using an aspect in HisExt$_1$. We will correct its behavior for the singular point $x = 0$ using HerExt$_2$.

The code in Listing 7, written in HisExt$_1$, advises the base code for drastically improving its efficiency and accuracy. Four pieces of advice are used. The first around advice (lines 202–204) intercepts executions of the procedure `Improve` and instead applies Newton's method:

$$a_{n+1} = \frac{1}{2}\left(a_n + \frac{x}{a_n}\right)$$

The second around advice (lines 205–207) intercepts `IsPrecise-Enough?` executions and checks instead whether or not

$$\left|(a_n)^2 - x\right| < \epsilon$$

where $\epsilon = \frac{1}{1000}$ is set in the third around advice (lines 208–211). The successive approximations now converge quadratically.

Running `main` and calling

```
call SQRT(5)
```

returns [7]

```
(num-val 161/72)
```

meaning $\frac{161}{72} = 2.2361111 = \sqrt{5.0001929} \doteq \sqrt{5}$.

The improved program works well for all non-negative inputs to SQRT, except for when the radicand is 0. In this case, `Improve` is called with the first argument a_n set to 0. The execution of `Improve` triggers the advice around `Improve` execution which divides x by a_n. Since the value of a_n is 0 an exception occurs.

3.3 Third-party Composition

The main point of this example is that HisExt$_1$ and HerExt$_2$ are subject to third-party composition with MyBase and work collaboratively:

- **Units of independent production.** HisExt$_1$ and HerExt$_2$ are independently constructed.
- **Units of composition.** MyBase, HisExt$_1$, and HerExt$_2$ are units of composition. MyBase can be used by itself (running only Listing 6). MyBase can be used with HisExt$_1$ alone (omitting Listing 8). MyBase can be used with HerExt$_2$ alone (omitting Listing 7). MyBase can be used with both HisExt$_1$ and HerExt$_2$.

[7] The result shown is the actual value returned by the Scheme [37] implementation.

- **Units of collaboration.** When HisExt_1 and HerExt_2 are both used they collaborate. In the absence of HerExt_2, calling

```
call SQRT(0)
```

results in

```
Error in /: undefined for 0.
```

However, when HerExt_2 with the advice code in Listing 8 are present, the correct value 0 is returned. The violating primitive division application is introduced by the advice of HisExt_1, yet intercepted by the advice of HerExt_2. This desired behavior is non-trivial because HisExt_1 was constructed without any prior knowledge of HerExt_2.

3.4 Analysis

In order to achieve a correct collaboration:

- The aspectual effect of all extension programs needs to be exposed to all the collaborating aspect mechanisms.
- Each individual aspect mechanism must hide its implementation from other aspect mechanisms.

3.4.1 *Effect Exposure*

In the context of multiple distinct aspect mechanisms, certain elements of the aspect program should be exposed to all collaborating aspect mechanisms. We call these elements the *aspectual effect*. The aspectual effect of an aspect program generally specifies the implementation of a crosscutting concern. We assume that the aspectual effect is expressed in the base language.

In our example, the aspectual effect of an $aspect_1 \in \mathsf{HisExt}_1$ is specified by advice-body expressions; the aspectual effect of an $aspect_2 \in \mathsf{HerExt}_2$ is specified by handler expressions. When HisExt_1 and HerExt_2 are composed together, their mechanisms must reflect each other's effect. Specifically, HisExt_1 aspects must be able to advise procedure calls made from the HerExt_2 handler expressions; and HerExt_2 handlers must be able to intercept exceptions introduced by the HisExt_1 pieces of advice.

3.4.2 *Implementation Hiding*

An aspect extension extends the base language with new functionality. For example, HisExt_1 adds advice binding, and HerExt_2 adds exception handling to the base language. An aspect mechanism that implements the new functionality must hide its internal operations from the other aspect mechanisms. In our example, pointcut matching and advice selection operations of the HisExt_1 mechanism must be hidden from the HerExt_2 mechanism. Conversely, testing whether the second division primitive argument evaluates to zero and the exception handler selection of HerExt_2 should be invisible to the HisExt_1 mechanism.

4. OUR APPROACH

Now that we have illustrated a desired behavior, we explain our solution to the aspect mechanism composition problem in general.

4.1 Aspect Mechanisms as Mixins

The primary idea is to view an aspect mechanism that extends a base mechanism as a *mixin* [11] that is applied to the base mechanism description. A description of a mechanism is an encoding of its implementation (e.g., a configuration of an abstract machine or its semantics). An *aspect mixin mechanism* transforms some of the base mechanism description and introduces some additional description.[8]

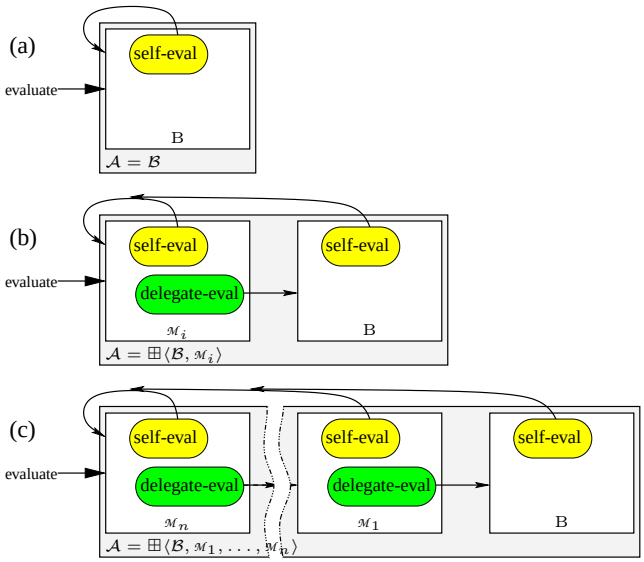

Figure 4: Mixin-like composition of aspect mechanisms: (a) design of a base mechanism; (b) design of an aspect mechanism; (c) third-party composition of aspect mechanisms.

By keeping a clean separation between the descriptions of the base and aspect mechanisms, the aspect mixin mechanism may be composed with other mechanisms that extend the same base language. The particular composition strategy may differ. In the next section, we show a concrete instance of this general approach.

4.2 Solution Instance

We illustrate the approach specifically for expression evaluation semantics. In our solution, the base mechanism $\mathcal{B}$ and the aspect mechanisms $\mathcal{M}_1, \ldots, \mathcal{M}_n$ compose into an AOP interpreter $\mathcal{A}$ (Figure 4). When the set of aspect mechanisms is empty, $\mathcal{A}$ behaves as a base interpreter (Figure 4(a)).

Each aspect mechanism $\mathcal{M}_i$ is *designed* as a wrapper around the base mechanism (Figure 4(b)). In the composition, $\mathcal{M}_i$ overrides the base functionality gracefully: the mechanism delegates all base operations to $\mathcal{B}$; it implements only its respective aspectual functionality.

To build a multi mechanism $\mathcal{A}$, the aspect mechanisms are subject to third-party composition (Figure 4(c)). The composed mechanisms are organized in a chain-of-responsibility [19], pipe-and-filter architecture [38]. Each aspect mechanism performs some part of the evaluation and forwards other parts of the evaluation to the next mechanism using delegation semantics ("super"-like calls) [6]. If an expression is delegated by all mechanisms then it is eventually evaluated in $\mathcal{B}$. All the mechanisms defer to $\mathcal{A}$ for the evaluation of recursive and other "self"-calls.

[8]We generally assume that a granularity requirement of an aspect mechanism can always be satisfied by either taking the most fine-grained description form (e.g., small-step operational semantics), or refining the current description (e.g., via annotations).

A subtlety in designing a collaborative aspect mechanism is deciding what to hide, what to delegate, and what to expose. A mechanism may hide its effect by reducing an expression internally. A mechanism may refine the next mechanism's semantics by delegating the evaluation. A mechanism may expose its effect by evaluating expressions in $\mathcal{A}$. The latter allows what is known as "weaving." The exposed expressions are then evaluated collaboratively by all the mechanisms. As a result, an effect of an aspect mechanism is made visible to all the other mechanisms. Hence, the mechanisms reflect one another's effect. Overall, a mechanism is considered a collaborative unit provided it properly hides, delegates, or exposes the evaluation.

Notation. The following notations are pertinent. We express functions in Curried form. The Curried function definition

$$\textbf{fn}\ pat_1\ pat_2 \ldots\ pat_n \Rightarrow exp$$

is the same as the lambda expression $\lambda\, pat_1.\lambda\, pat_2. \ldots .\lambda\, pat_n.exp$. Correspondingly, we write a list of function arguments with no parentheses or commas to express a function application that takes the first argument as its single parameter, which could be a tuple, constructs and returns a new function, which then takes the next argument as its single parameter, and so on. In function types, '$\rightarrow$' associates to the right.

We use the form $(id\ \textbf{as}\ pat)$ in a formal argument to bind an identifier id to a value and match the value with a pattern pat. Variables in the pattern bind to their corresponding values. We use **val** $pat = val$ to split apart a value. The symbol '_' stands for an anonymous variable (don't care). The symbol '$\diamond$' denotes an empty mapping and '[] ' denotes an empty list.

4.2.1 Expressions

The Base grammar introduces a set of expression productions; $\mathbf{Exp}_0$ is the set of base expressions whose pattern matches one of these productions. Each of the extensions $\mathsf{Ext}_1, \mathsf{Ext}_2, \ldots, \mathsf{Ext}_n$ may extend Base with its own respective set of additional expressions $\mathbf{Exp}_1, \mathbf{Exp}_2, \ldots, \mathbf{Exp}_n$.[9] The set $\mathbf{Exp}_A$ of AOP expressions is hence a union of pairwise disjointed expression sets defined by:

$$\mathbf{Exp}_A = \mathbf{Exp}_0 + \mathbf{Exp}_1 + \mathbf{Exp}_2 + \cdots + \mathbf{Exp}_n$$

Note that in an extended grammar, an expression in $\mathbf{Exp}_0$ may contain subexpressions not in $\mathbf{Exp}_0$. For example, in the case of MyBase and HisExt_1, the expression

```
*(2,proceed)
```

is a base expression (because its pattern matches the production `primapp-exp` in MyBase) but **proceed** is not.

4.2.2 Overall Semantics

Let $\mathcal{A}[\![exp]\!]$ denote the meaning of an AOP expression exp. Our goal is to be able to build the multi mechanism $\mathcal{A}$ by composing the base mechanism $\mathcal{B}$ and the mutually independent aspect mechanisms $\mathcal{M}_1, \ldots, \mathcal{M}_n$. We use the term *AOP configuration* to denote the state of a multi mechanism $\mathcal{A}$. An AOP configuration $cfg \in \mathbf{Cfg}_A$ is a vector of configurations of the composed mechanisms:

$$\mathbf{Cfg}_A = \mathbf{Cfg}_0 \times \mathbf{Cfg}_1 \times \mathbf{Cfg}_2 \times \cdots \times \mathbf{Cfg}_n$$

[9] We assume that $\mathbf{Exp}_i \cap \mathbf{Exp}_j = \phi$ for all $0 \leq i < j \leq n$.

where $\mathbf{Cfg}_0$ denotes a domain of the base mechanism states, and $\mathbf{Cfg}_i, 1 \leq i \leq n$, denotes a domain of the aspect mechanism $\mathcal{M}_i$ states. For example, a MyBase mechanism configuration comprises a procedure environment, a variable environment, and a store. A HisExt_1 mechanism configuration comprises a list of advice, a "current" join point, and a "current" proceed computation.

The effect of evaluating an expression $exp \in \mathbf{Exp}_A$ is to change the AOP configuration. The meaning of an expression $exp \in \mathbf{Exp}_A$, denoted $\mathcal{A}[\![exp]\!]$, is defined to be a partial function on configurations:

$$\mathcal{A} : \mathbf{Exp}_A \rightarrow \overbrace{(\mathbf{Cfg}_A \hookrightarrow \mathbf{Cfg}_A)}^{\mathbf{Cont}_A}$$

We denote by $\mathbf{Cont}_A$ the set of partial functions on $\mathbf{Cfg}_A$.

4.2.3 Design Guidelines for the Base Mechanism

$\mathcal{B}$ provides semantics for expressions in Base. The meaning of an expression $exp \in \mathbf{Exp}_0$ in Base, denoted $\mathcal{B}[\![exp]\!]$, is expected to be defined as:

$$\mathcal{B} : \mathbf{Exp}_0 \rightarrow \mathbf{Cont}_A$$

The semantical function $\mathcal{B}$ should adhere to the following design principles:

- All sub-reductions within a $\mathcal{B}$-reduction are reduced by calling $\mathcal{A}$ instead of $\mathcal{B}$.
- $\mathcal{B}$ only accesses and updates the head $\mathbf{Cfg}_0$-element of the $cfg \in \mathbf{Cfg}_A$ configuration list, and carries the tail through the computation.

Note that the fact that $\mathcal{B}$ is defined in terms of $\mathbf{Cfg}_A$ does not mean that $\mathcal{A}$ or n are known at the time of writing $\mathcal{B}$. At the time of writing the base mechanism, $\mathcal{A}$ is assumed to delegate everything to $\mathcal{B}$:

$$\mathcal{A}[\![exp]\!] = \begin{cases} \mathcal{B}[\![exp]\!] & exp \in \mathbf{Exp}_0 \\ \bot & \text{otherwise} \end{cases}$$

where $\bot$ stands for "undefined." Let $\hat{\mathcal{B}} : \mathbf{Exp}_0 \rightarrow \mathbf{Cfg}_0 \hookrightarrow \mathbf{Cfg}_0$ denote the evaluation semantics for Base with its standard signature. $\mathcal{B}$ has a different signature than $\hat{\mathcal{B}}$ but the same behavior as $\hat{\mathcal{B}}$. $\forall exp \in \mathbf{Exp}_0, \forall cfg = cfg_0 :: cfg^* \in \mathbf{Cfg}_A$:

$$\mathcal{B}[\![exp]\!]\ cfg = \begin{cases} cfg_0' :: cfg^* & \hat{\mathcal{B}}[\![exp]\!]\ cfg_0 = cfg_0' \\ \bot & \hat{\mathcal{B}}[\![exp]\!]\ cfg_0 = \bot \end{cases}$$

4.2.4 Design Guidelines for an Aspect Mechanism

We construct the aspect mechanism $\mathcal{M}_i$ for an aspect extension Ext_i as the override combination[10] of a semantics transformer $\mathcal{T}_i$ and a semantical function $\mathcal{E}_i$:

$$\textbf{val}\ \mathcal{M}_i = \quad \textbf{fn}\ eval \Rightarrow (\mathcal{T}_i\ eval\ ) \oplus \mathcal{E}_i$$

Semantics for the Ext_i's newly introduced expressions $\mathbf{Exp}_i$ is defined by:

$$\mathcal{E}_i : \mathbf{Exp}_i \rightarrow \mathbf{Cont}_A$$

[10] For two partial functions g and h, their override combination $g \oplus h$ (h overrides g), is defined by:

$$(g \oplus h)(x) = \begin{cases} h(x) & x \in \mathbf{dom}\ h \\ g(x) & otherwise \end{cases}$$

The introduction of Ext_i into the base language also requires a change to the evaluation semantics for a non-empty[11] subset of the existing base language expressions $\mathbf{Exp}_0^i \subseteq \mathbf{Exp}_0$. We define this part of the semantics for Ext_i as a language semantics transformer:

$$\mathcal{T}_i : \overbrace{(\mathbf{Exp}_0 \rightarrow \mathbf{Cont}_A)}^{\mathbf{Eval}_0} \rightarrow \overbrace{(\mathbf{Exp}_0^i \rightarrow \mathbf{Cont}_A)}^{\mathbf{Eval}_0^i}$$

The semantics transformer $\mathcal{T}_i$ should adhere to the following design principles:

- $\mathcal{T}_i$ defines the semantics for Ext_i and nothing more. Let $\mathcal{B}'$ denote a semantical function with the same signature as $\mathcal{B}$ or an extended signature.[12] $\mathcal{T}_i(\mathcal{B}')$ delegates the evaluation to $\mathcal{B}'$ whenever the base language semantics is required.
- $\mathcal{T}_i(\mathcal{B}')$ accesses only the $\mathbf{Cfg}_0$- and $\mathbf{Cfg}_i$-elements in a $cfg \in \mathbf{Cfg}_A$ configuration, while the rest are carried through the computation.

Note that allowing the aspect mechanism access to the $\mathbf{Cfg}_0$ element is needed for modeling interesting cases of aspect mechanism interactions.

4.2.5 *Third-party Construction of an AOP Language*

Let $\mathbf{K} = \{k_i\}_{i=1}^{l}$ be an ordered index set, and let $\mathcal{M}_{k_1}, \ldots, \mathcal{M}_{k_l}$ denote the $l \leq n$ aspect mechanisms to be composed. Let $\mathcal{B}$ denote the Base mechanism, and let $\mathcal{A}^{\mathbf{K}}$ denote the multi mechanism being constructed.

We construct the multi mechanism $\mathcal{A}^{\mathbf{K}}$ as the composition:

$$\mathcal{A}^{\mathbf{K}} = \mathcal{A}_l = \boxplus\langle \mathcal{B}, \mathcal{M}_{k_1}, \ldots, \mathcal{M}_{k_l} \rangle$$

where the composition semantics for $\boxplus$ is defined as follows. The meaning of $exp \in \mathbf{Exp}_A$, denoted $\mathcal{A}_l[\![exp]\!]$, is given by the recurrence relation:

$$\begin{aligned} \mathcal{A}_0 &= \mathcal{B} \\ \mathcal{A}_l &= \mathcal{A}_{l-1} \oplus (\mathcal{M}_{k_l}\ \mathcal{A}_{l-1}) \end{aligned}$$

By construction,

$$\mathcal{A}_l : \left(\mathbf{Exp}_0 + \mathbf{Exp}_{k_1} + \cdots + \mathbf{Exp}_{k_l}\right) \rightarrow \mathbf{Cont}_A$$

is of the right signature and obeys the composition principle. To illustrate the construction, we conclude by elaborating the first three instances:

- For $l = 0$, we have that $\mathbf{Exp}_A = \mathbf{Exp}_0$, and the meaning of $exp \in \mathbf{Exp}_A$ is the same as the meaning of exp in Base:

$$\mathcal{A}^{\phi} : \mathbf{Exp}_0 \rightarrow \mathbf{Cont}_A$$

$$\mathcal{A}^{\phi}[\![exp]\!] = \mathcal{B}[\![exp]\!]$$

- For $l = 1$ and the singleton index set $\{i\}$ for some $1 \leq i \leq n$, we have that $\mathbf{Exp}_A = \mathbf{Exp}_0 + \mathbf{Exp}_i$. The meaning of $exp \in \mathbf{Exp}_A$ is

$$\mathcal{A}^{\{i\}} : (\mathbf{Exp}_0 + \mathbf{Exp}_i) \rightarrow \mathbf{Cont}_A$$

[11] W.l.o.g., assume $\mathbf{Exp}_0^i \neq \phi$.

[12] An extended $\mathcal{B}$ may have a signature $\mathcal{B}' : \mathbf{Exp}_0' \rightarrow \mathbf{Cont}_A$, where $\mathbf{Exp}_0' \supseteq \mathbf{Exp}_0$.

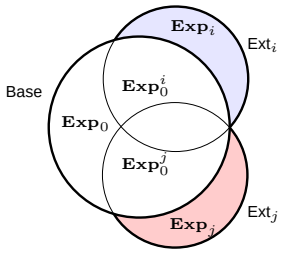

Figure 5: Expression domains for $l = 2$

We construct:

$$\mathcal{A}^{\{i\}} = \mathcal{B} \oplus \overbrace{(\mathcal{T}_i\ \mathcal{B}) \oplus \mathcal{E}_i}^{\mathcal{M}_i\ \mathcal{B}}$$

$$\mathcal{A}^{\{i\}}[\![exp]\!] = \begin{cases} \mathcal{E}_i[\![exp]\!] & exp \in \mathbf{Exp}_i \\ (\mathcal{T}_i\ \mathcal{B})[\![exp]\!] & exp \in \mathbf{Exp}_0^i \\ \mathcal{B}[\![exp]\!] & otherwise \end{cases}$$

- For $l = 2$ and the ordered index set $\{i, j\}$ for some $1 \leq i, j \leq n$, we have that $\mathbf{Exp}_A = \mathbf{Exp}_0 + \mathbf{Exp}_i + \mathbf{Exp}_j$ (Figure 5). The meaning of $exp \in \mathbf{Exp}_A$ is

$$\mathcal{A}^{\{i,j\}} : (\mathbf{Exp}_0 + \mathbf{Exp}_i + \mathbf{Exp}_j) \rightarrow \mathbf{Cont}_A$$

We construct:

$$\mathcal{A}^{\{i,j\}} = \mathcal{A}^{\{i\}} \oplus \overbrace{(\mathcal{T}_j\ \mathcal{A}^{\{i\}}) \oplus \mathcal{E}_j}^{\mathcal{M}_j\ \mathcal{A}^{\{i\}}}$$

$$\mathcal{A}^{\{i,j\}}[\![exp]\!] = \begin{cases} \mathcal{E}_j[\![exp]\!] & exp \in \mathbf{Exp}_j \\ \mathcal{E}_i[\![exp]\!] & exp \in \mathbf{Exp}_i \\ (\mathcal{T}_j\ \mathcal{B})[\![exp]\!] & exp \in \mathbf{Exp}_0^j - \mathbf{Exp}_0^i \\ (\mathcal{T}_i\ \mathcal{B})[\![exp]\!] & exp \in \mathbf{Exp}_0^i - \mathbf{Exp}_0^j \\ (\mathcal{T}_j\ (\mathcal{T}_i\ \mathcal{B}))[\![exp]\!] & exp \in \mathbf{Exp}_0^i \cap \mathbf{Exp}_0^j \\ \mathcal{B}[\![exp]\!] & otherwise \end{cases}$$

5. IMPLEMENTATION

As a proof of concept we have implemented MyBase, HisExt_1, and HerExt_2 for the example presented in Section 3. This section provides the implementation details more formally to the so-inclined reader.

5.1 Base Mechanism Implementation

The domain $\mathbf{Exp}_A$ of AOP expressions includes MyBase, HisExt_1, and HerExt_2 expressions. We define $\mathbf{Exp}_0$ by extending the MyBase expression grammar $\mathbf{Exps}$ (Figure 1) with a set of annotated expression *annotated-exp* (Figure 6).

$$\mathbf{Exp}_0 = \mathbf{Exps} + \textit{annotated-exp}$$

Annotated expressions extend the interface of the base mechanism to satisfy granularity needs of the HisExt_1 and HerExt_2 mechanisms. In the extended grammar, a complex expression (Figure 7) includes annotated expressions as subexpressions.

The base configuration domain $\mathbf{Cfg}_0$ consist of a procedure environment domain $\mathbf{Env}_{\mathrm{P}}$, a variable environment domain $\mathbf{Env}_{\mathrm{V}}$,

annotated-exp	=	*procbody-exp* | *procarg-exp* | *primarg-exp* | *assignrhs-exp* | *block-exp* | *letbody-exp* | *letrhs-exp* | *if-exp* | *then-exp* | *else-exp*	
procbody-exp	=	$\mathbf{Exps} \times \mathbf{PNm}$	Procedure body
procarg-exp	=	$\mathbf{Exps} \times (\mathbf{PNm} \times \mathbf{Var})$	Procedure arg
primarg-exp	=	$\mathbf{Exps} \times (Prim \times Int)$	Primitive arg
assignrhs-exp	=	$\mathbf{Exps} \times \mathbf{Var}$	Assignment RHS
block-exp	=	$\mathbf{Exps} \times Int$	Block element
letbody-exp	=	$\mathbf{Exps} \times \mathbf{Var}^*$	Let body
letrhs-exp	=	$\mathbf{Exps} \times (\mathbf{Var} \times Int)$	Let env RHS
if-exp	=	$\mathbf{Exps} \times \{\texttt{if}\}$	If exp
then-exp	=	$\mathbf{Exps} \times \{\texttt{then}\}$	Then exp
else-exp	=	$\mathbf{Exps} \times \{\texttt{else}\}$	Else exp

Figure 6: Annotated expressions

app-exp	=	$\mathbf{PNm} \times$ *procarg-exp**	Procedure call
begin-exp	=	*block-exp**	Block
cond-exp	=	*if-exp* $\times$ *then-exp* $\times$ *else-exp*	Conditional exp
assign-exp	=	$\mathbf{Var} \times$ *assignrhs-exp*	Assignment
let-exp	=	$\mathbf{Var}^* \times$ *letrhs-exp*$^* \times$ *letbody-exp*	Let
primapp-exp	=	$Prim \times$ *primarg-exp**	Primitive app

Figure 7: Complex expressions

$cfg_0 \in \mathbf{Cfg}_0$	=	$\mathbf{Env}_\mathrm{P} \times \mathbf{Env}_\mathrm{V} \times \mathbf{Store}$	Base configuration
$env_V \in \mathbf{Env}_\mathrm{V}$	=	$\mathbf{Var} \to \mathbf{Loc}$	Variable envs
$sto \in \mathbf{Store}$	=	$\mathbf{Loc} \to \mathbf{Val}$	Value Stores
$env_P \in \mathbf{Env}_\mathrm{P}$	=	$\mathbf{PNm} \to \mathbf{Proc}$	Procedure envs
$\theta \in \mathbf{Proc}$	=	$\mathbf{Var}^* \times$ *procbody-exp*	Procedures

Figure 8: MyBase domains

and a value store domain **Store** (Figure 8). A procedure is represented as a closure that contains argument names and a procedure body expression. The other definitions are omitted.

The evaluation semantics $\mathcal{B}$ (Figure 9) for $\mathbf{Exp}_0$ expressions satisfies the design principles for the base mechanisms: (1) all expression evaluations in $\mathcal{B}$ are *exposed* to $\mathcal{A}$ (highlighted in the figure); (2) it accesses and updates only the $\mathbf{Cfg}_0$-element of the configuration; (3) the other configurations are carried through the computation.

5.2 Aspect Mechanism Implementation

The semantics for Ext_i is specified using three constructor functions:

- *build-*$\mathcal{E}_i$ constructs an evaluator for $\mathbf{Exp}_i$ expressions:

$$\textit{build-}\mathcal{E}_i : Int \to (\mathbf{Exp}_i \to \mathbf{Cont}_A)$$

- *build-*$\mathcal{T}_i$ constructs the semantics transformer for the Ext_i:

$$\textit{build-}\mathcal{T}_i : Int \to \overbrace{(\mathbf{Exp}_0 \to \mathbf{Cont}_A) \to (\mathbf{Exp}_0^i \to \mathbf{Cont}_A)}^{\mathbf{Eval}_0 \to \mathbf{Eval}_0^i}$$

$$\begin{array}{l}
\textbf{val}\ \mathcal{B} : \mathbf{Exp}_0 \to \mathbf{Cont}_A \\
\quad = \textbf{fn}\ (\texttt{lit-exp}\ \langle num \rangle)\ \langle _, _, sto \rangle :: cfg^* \Rightarrow \\
\qquad \langle \diamond, \diamond, sto[0 \mapsto (\textit{num-val num})] \rangle :: cfg^* \\
\quad \mid\ \textbf{fn}\ (\texttt{true-exp}\ \langle\rangle)\ \langle _, _, sto \rangle :: cfg^* \Rightarrow \\
\qquad \langle \diamond, \diamond, sto[0 \mapsto (\textit{bool-val}\ \texttt{\#t})] \rangle :: cfg^* \\
\quad \mid\ \textbf{fn}\ (\texttt{false-exp}\ \langle\rangle)\ \langle _, _, sto \rangle :: cfg^* \Rightarrow \\
\qquad \langle \diamond, \diamond, sto[0 \mapsto (\textit{bool-val}\ \texttt{\#f})] \rangle :: cfg^* \\
\quad \mid\ \textbf{fn}\ (\texttt{app-exp}\ \langle pname, [exp_1, \ldots, exp_n] \rangle)\ cfg_0 :: cfg^* \\
\qquad \textbf{let} \\
\qquad\quad \textbf{val}\ \langle env_P, env_V, sto \rangle = cfg_0 \\
\qquad\quad \textbf{val}\ \langle [id_1, \ldots, id_n], exp_{\mathrm{proc}} \rangle = env_P\ pname \\
\qquad\quad \textbf{val}\ \langle _, _, sto_1 \rangle :: cfg_1^* = \\
\qquad\qquad \mathcal{A}\ exp_1\ \langle env_P, env_V, sto \rangle :: cfg^* \\
\qquad\quad \textbf{val}\ v_1 = sto_1\ 0 \\
\qquad\quad \ldots \\
\qquad\quad \textbf{val}\ \langle _, _, sto_n \rangle :: cfg_n^* = \\
\qquad\qquad \mathcal{A}\ exp_n\ \langle env_P, env_V, sto_{n-1} \rangle :: cfg_{n-1}^* \\
\qquad\quad \textbf{val}\ v_n = sto_n\ 0 \\
\qquad\quad \textbf{val}\ sto_{n+1} = sto_n[l_1 \mapsto v_1],\ l_1 \notin \mathbf{dom}\ sto_n \\
\qquad\quad \ldots \\
\qquad\quad \textbf{val}\ sto_{2n} = sto_{2n-1}[l_n \mapsto v_n],\ l_n \notin \mathbf{dom}\ sto_{2n-1} \\
\qquad \textbf{in} \\
\qquad\quad \mathcal{A}\ exp_{\mathrm{proc}}\ \langle env_P, \diamond[id_1 \mapsto l_1, \ldots, id_n \mapsto l_n], sto_{2n} \rangle :: cfg_n^* \\
\qquad \textbf{end} \\
\quad \mid\ \ldots \\
\quad \mid\ \textbf{fn}\ (\texttt{annotated-exp}\ \langle exp, _ \rangle)\ cfg \Rightarrow \mathcal{A}\ exp\ cfg
\end{array}$$

Figure 9: MyBase semantical function

- *build-*$\mathcal{M}_i$ constructs the aspect mixin mechanism $\mathcal{M}_i$ for Ext_i:

$$\begin{array}{l}
\textbf{val}\ \textit{build-}\mathcal{M}_i : Int \to \mathbf{Eval}_0 \to (\mathbf{Exp}_0^i + \mathbf{Exp}_i) \to \mathbf{Cont}_A \\
\quad = \quad \textbf{fn}\ pos\ eval \Rightarrow (\textit{build-}\mathcal{T}_i\ pos\ eval) \oplus (\textit{build-}\mathcal{E}_i\ pos)
\end{array}$$

The Int arguments provides the position of the extension's configuration domain $\mathbf{Cfg}_i$ within $\mathbf{Cfg}_A$.

Intuitively, the aspect mechanisms are implemented as mixins to the base mechanism (Figure 10). The HisExt_1 mechanism $\mathcal{M}_1$ refines the semantics for `app-exp` and `procbody-exp` and introduces semantics for `proceed-exp`. The HerExt_2 mechanism $\mathcal{M}_2$ refines `primarg-exp` and `procbody-exp` and introduces semantics for `raise-exp`.

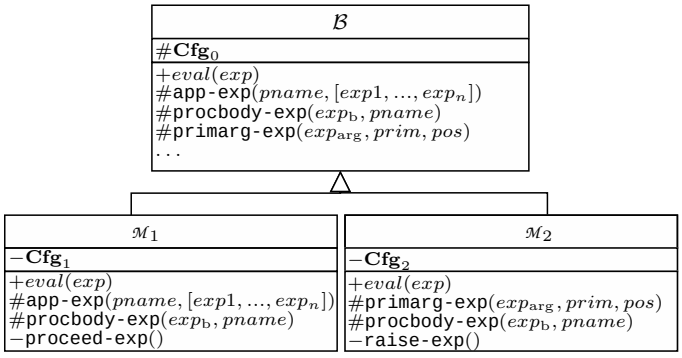

Figure 10: Aspect mechanisms as mixins

$exp \in \mathbf{Exp}_{\text{adv}}$	$= \mathbf{Exp}_0 + \mathbf{Exp}_1$	Advice exps
$cfg_1 \in \mathbf{Cfg}_1$	$= \mathbf{Adv}^* \times \mathbf{JP} \times \mathbf{Cont}_A$	Configuration
$adv \in \mathbf{Adv}$	$= \mathbf{PCD} \times \mathbf{Exp}_{\text{adv}}$	Advice
$jp \in \mathbf{JP}$	$= \{\texttt{call}, \texttt{exec}\} \times \mathbf{PNm} \times$ $\mathbf{Var}^* \times \mathbf{Val}^* \times \mathbf{JP} + \mathbf{Unit}$	Join points
$pcd \in \mathbf{PCD}$		Pointcuts
$effect \in \mathbf{Effect}$	$= \mathbf{Bnd}^* \times \mathbf{Exp}_{\text{adv}}$	Effects
$bnd \in \mathbf{Bnd}$	$= \mathbf{Var} \times \mathbf{Val}$	Binding

Figure 11: HisExt$_1$ domains

5.2.1 HisExt$_1$ *Mechanism*

The aspect mechanism $\mathcal{M}_1$ transforms the semantics for procedure calls and executions, and supplies semantics for $\mathbf{Exp}_1$'s new proceed expression:

$$\mathbf{Exp}_0^1 = \{\textit{app-exp}, \textit{procbody-exp}\}$$
$$\mathbf{Exp}_1 = \{\textit{proceed-exp}\}$$

A configuration $cfg_1 \in \mathbf{Cfg}_1$ for HisExt$_1$ (Figure 11) comprises a set of advice, a "current" join point, and a "current" proceed continuation. An advice $adv \in \mathbf{Adv}$ is derived directly from HisExt$_1$'s syntax. A join point $jp \in \mathbf{JP}$ is an abstraction of the procedure call stack. It stores the name and formal and actual arguments of a corresponding procedure. The third element provides a meaning for proceed expressions. The effect and binding domains are internal to the mechanism. An effect carries a set of bindings and an advice body expression. The bindings provide an appropriate variable environment for evaluating the advice body expression.

The interesting part of the aspect mechanism $\mathcal{M}_1$ implementation is given by *build-*$\mathcal{T}_1$ (Figure 12). *build-*$\mathcal{T}_1$ defines a transformer of the semantics for procedure calls and procedure executions. The new semantics creates a join point, matches it against an advice list, and applies selected advice effects in *app-eff*. The function ensures that the mechanism's configuration properly reflects a "current" join point by setting it before and after an effect application.

app-eff has two general behaviors. If the effect list is empty then the expression evaluation is *delegated*. Otherwise, the function *exposes* the effect by evaluating the advice expression exp_{adv} in $\mathcal{A}$. exp_{adv} is evaluated in a properly constructed variable environment $env_V{}'$ and a proceed continuation $procd'$.

app-eff ensures that the mechanism configuration always stores a proper proceed continuation in the same manner as *build-*$\mathcal{T}_1$ reflects a "current" join point. This makes *build-*$\mathcal{E}_1$ straightforward (Figure 13). The meaning of a `proceed-exp` expression is given by the proceed continuation obtained from the configuration. The continuation then runs *app-eff* on the rest of the effect list. In other words, a `proceed-exp` expression either evaluates the next advice in $\mathcal{A}$ or delegates the evaluation to *eval* if there is no advice left.

Due to space considerations, we omit the HisExt$_1$ functions *match-jp*, *build-jp* and *build-adv-env*, which do not affect the mechanism composition semantics.

```
local
 val app-eff : Int → Effect* → Eval0 → Eval0
  = fn _ [] eval exp cfg ⇒ eval exp cfg
  | fn i ⟨bnd*adv, expadv⟩ :: effect* eval ⇒
     fn exp ⟨envP, envV, sto⟩ :: cfg* ⇒
      let
       val ⟨adv*, jp, procd⟩ = πi(cfg*)
       val procd' : Cont_A
        = fn ⟨_, _, sto'⟩ :: cfg*' ⇒
          app-eff i effect* eval exp ⟨envP, envV, sto'⟩ :: cfg*'
       val ⟨envV', sto'⟩ = build-adv-env bnd*adv sto
       val cfg*' = cfg*[i ↦ ⟨adv*, jp, procd'⟩]
       val cfg'0 :: cfg*'' = A expadv ⟨envP, envV', sto'⟩ :: cfg*'
      in
       cfg'0 :: cfg*''[i ↦ ⟨adv*, jp, procd⟩]
      end
 ...
in
 val build-T1 : Int → Eval0 → Eval1_0
 = fn i eval exp cfg0 :: cfg* ⇒
 let
  val ⟨adv*, jpenc, procd⟩ = πi(cfg*)
  val jp = build-jp exp jpenc cfg0
  val effect* = match-jp jp adv*
  val cfg*' = cfg*[i ↦ ⟨adv*, jp, procd⟩]
  val cfg'0 :: cfg*'' = app-eff i effect* eval exp cfg0 :: cfg*'
 in
  cfg'0 :: cfg*''[i ↦ ⟨adv*, jpenc, procd⟩]
 end
end
```

Figure 12: *build-*$\mathcal{T}_1$

```
val build-E1 : Int → Exp1 → Cont_A
= fn i (proceed-exp ⟨⟩) (cfg as _ :: cfg*) ⇒
let
 val ⟨_, _, procd⟩ = πi(cfg*)
in
 procd cfg
end
```

Figure 13: *build-*$\mathcal{E}_1$

5.2.2 HerExt$_2$ *Mechanism*

The $\mathcal{M}_2$ mechanism for HerExt$_2$ transforms the semantics for a primitive argument and procedure execution expressions, and supplies semantics for $\mathbf{Exp}_2$'s new raise expression:

$$\mathbf{Exp}_0^2 = \{\textit{primarg-exp}, \textit{procbody-exp}\}$$
$$\mathbf{Exp}_2 = \{\textit{raise-exp}\}$$

A configuration $cfg_2 \in \mathbf{Cfg}_2$ (Figure 14) stores a list of handlers, a stack of currently executing procedures (a list of procedure names), and a "current" raise continuation. A handler $hnd \in \mathbf{Handler}$ is derived from the syntax of HerExt$_2$. It contains a name of a guarded procedure and a handler expression. A handler expression may contain a `raise-exp` expression.

$exp \in \mathbf{Exp}_{\text{hnd}}$	$=$	$\mathbf{Exp}_0 + \mathbf{Exp}_2$	Handler exps
$cfg_2 \in \mathbf{Cfg}_2$	$=$	$\mathbf{Handler}^* \times \mathbf{PNm}^* \times \mathbf{Cont}_A$	Configuration
$hnd \in \mathbf{Handler}$	$=$	$\mathbf{PNm} \times \mathbf{Exp}_{\text{hnd}}$	Handlers

Figure 14: HerExt$_2$ domains

```
local
 val app-handler : Int → Exp*hnd → Cont_A
  = fn _ [] cfg ⇒ cfg
  | fn i exp::exp* ⟨env_P, _, sto⟩::cfg* ⇒
     let
      val ⟨hnd*, stack, raise⟩ = π_i(cfg*)
      val v = sto 0
      val raise' : Cont_A
       = fn ⟨env_P, env_V, sto⟩::cfg* ⇒
         app-handler exp* ⟨env_P, env_V, sto[0 ↦ v]⟩::cfg*
      val cfg*' = cfg*[i ↦ ⟨hnd*, stack, raise'⟩]
      val cfg'_0::cfg*'' = A exp ⟨env_P, ◇, sto⟩::cfg*'
     in
      cfg'_0::cfg*''[i ↦ ⟨hnd*, stack, raise⟩]
     end
 ...
in
 val build-T_2 : Int → Eval_0 → Eval_0^2
  = fn i eval (primarg-exp ⟨_, prim, pos⟩ as exp) cfg ⇒
     let
      val ⟨env_P, env_V, _⟩::_ = cfg
      val (cfg' as ⟨_, _, sto⟩::cfg*) = eval exp cfg
     in
      if (sto 0 = (num-val 0)) ∧ prim = "/" ∧ pos = 2)
       then
        let
         val ⟨hnd*, stack, _⟩ = π_i(cfg*)
         val exp*hnd = match-handler hnd* stack
        in
         app-handler i exp*hnd ⟨env_P, env_V, sto⟩::cfg*
        end
       else cfg'
     end
  | fn i eval (procbody-exp ⟨_, pname⟩ as exp) cfg ⇒
     let
      val cfg_0::cfg* = cfg
      val ⟨hnd*, stack, raise⟩ = π_i(cfg*)
      val cfg*' = cfg*[i ↦ ⟨hnd*, pname::stack, raise⟩]
      val cfg'_0::cfg*'' = eval exp cfg_0::cfg*'
     in
      cfg'_0::cfg*''[i ↦ ⟨hnd*, stack, raise⟩]
     end
end
```

Figure 15: *build-*$\mathcal{T}_2$

The new semantics for `primarg-exp` enables the invocation of a handler in an exceptional situation when the second argument of a division primitive evaluates to zero. In this case, *build-*$\mathcal{T}_2$ (Figure 15) selects a list of handler expressions using *match-handler* and invokes them using *app-handler*. If no exception occurs, the original semantics is used.

```
val build-E_2 : Int → Exp_2 → Cont_A
 = fn i (raise-exp ⟨⟩) (cfg as _::cfg*) ⇒
let
 val ⟨_, _, raise⟩ = π_i(cfg*)
in
 raise cfg
end
```

Figure 16: *build-*$\mathcal{E}_2$

The mechanism reflects the execution stack in its configuration by transforming the semantics for `procbody-exp` expressions. The new semantics simply pushes the stack before and pops it after applying *eval*.

app-handler produces a configuration transformer from a list of handler expressions. If the list is empty then the transformer is the identity function. Otherwise, the configuration is constructed by evaluating in $\mathcal{A}$ the first handler expression. The function also constructs and reflects a raise continuation in the mechanism configuration. The continuation simply applies *app-handler* to the rest of the handlers.

The *build-*$\mathcal{E}_2$ function (Figure 16) is similar to *build-*$\mathcal{E}_1$. The meaning of a `raise-exp` expression is provided by the raise continuation drawn from the configuration.

Due to space considerations, we omit the *match-handler* function of HerExt$_2$. This function bars no affect on the mechanism composition semantics.

5.3 Constructing an AOP Language

We construct the semantical function for the composed AOP language as follows:

$$\mathcal{A} = \boxplus\, \langle \mathcal{B}, \mathcal{M}_1, \mathcal{M}_2 \rangle$$

where

$$\mathcal{M}_1 = \textit{build-}\,\mathcal{M}_1\ 1$$

and

$$\mathcal{M}_2 = \textit{build-}\,\mathcal{M}_2\ 2$$

The meaning of a program

$$p = \langle base, aspect_1, aspect_2 \rangle$$

in the composed AOP language is defined as:

$$\mathfrak{M}[\![p]\!] = \mathcal{A}\ exp_{\text{main}}\ \langle cfg_0, cfg_1, cfg_2 \rangle$$

such that

exp_{main}	$= (\texttt{app-exp}\ \langle \text{`}main, [\,]\,\rangle)$	
cfg_0	$= \langle env_P, \diamond, \diamond \rangle$	$env_P = \mathfrak{D}_0[\![base]\!]$
cfg_1	$= \langle adv^*, \langle\rangle, \diamond \rangle$	$adv^* = \mathfrak{D}_1[\![aspect_1]\!]$
cfg_2	$= \langle hnd^*, [\,], \diamond \rangle$	$hnd^* = \mathfrak{D}_2[\![aspect_2]\!]$

6. DISCUSSION AND FUTURE WORK

Our study of constructing an AOP language with multiple aspect extensions opens interesting research questions.

6.1 Alternative Collaboration Semantics

The co-existence of multiple aspect extensions raises a question concerning the desired policy of collaboration. The solution instance we presented in Section 4.2 defines the combinator $\boxplus$ to "wrap" aspect mechanisms around the original meaning and around each other. We call this a composition with *wrapping* semantics.

Composition with wrapping semantics allows to compose arbitrary aspect mechanisms as long as the mechanisms can be defined as semantics transformers. However, wrapping semantics limits the ability of the multi mechanism to observe and affect the program execution. In this section we elaborate on the restrictions, and discuss how alternative solution instances can be constructed.

6.1.1 Observed Execution

Wrapping semantics grants the aspect mechanism with complete control over the original meaning and with the option to override the semantics. For example, the HisExt_1 mechanism might disable the original semantics of *app-exp* and *procbody-exp* expressions. A mechanism can either delegate the expression evaluation to the next mechanism or evaluate the expression itself. In the latter case, the evaluated expression is "filtered" out (hidden) from the aspect mechanisms downstream. However, when delegating is semantically not the right thing to do, hiding is unavoidable. For example, the HisExt_1 mechanism must filter out procedure calls and executions that are advised with no **proceed**.

Collaboration with wrapping semantics is therefore sensitive to the order of composition. The program example in Listing 9 illustrates a collaboration of HisExt_1 and HerExt_2 with wrapping semantics.

Listing 9: Collaboration with wrapping semantics

```
1 program {procedure main() { 1 } }
2 aop1 { around(): pexecution(main) {/(1,0)} }
3 aop2 { guard_cflow main resume_with 2 }
```

If the AOP language is constructed as

$$\mathcal{A} = \boxplus \langle \mathcal{B}, \mathcal{M}_2, \mathcal{M}_1 \rangle$$

$\mathcal{M}_1$ applies first and replaces the *procbody-exp* of **main** with the advice body expression. Consequently, the execution of **main** is not reflected in $\mathcal{M}_2$'s execution stack and $\mathcal{M}_2$ would not guard the division. The program would therefore throws a divide-by-zero exception. On the other hand, if the language is constructed as

$$\mathcal{A} = \boxplus \langle \mathcal{B}, \mathcal{M}_1, \mathcal{M}_2 \rangle$$

the exception is caught.

Generally, with wrapping semantics the various mechanisms observe a program execution differently. Only the first mechanism in $\mathcal{A}$ has a complete view of the execution. Downstream mechanisms view less of the execution than upstream ones.

Alternatively, one can provide a collaboration semantics where all the mechanisms share a complete, coherent view of the execution. This can be achieved by decoupling the reification and reflection processes in a mechanism. With such a semantics, every expression evaluated in $\mathcal{A}$ is reified by all the mechanisms. The evaluation semantics is then constructed by all the mechanisms collaboratively with respect to the ordering. Given this alternative semantics, the program example in Listing 9 would produce no exception independently of the ordering of $\mathcal{M}_1$ and $\mathcal{M}_2$.

6.1.2 Complex Compositions

Wrapping semantics does not support complex compositions of aspectual effects. The aspectual effects in different extensions "wrap" around each other when they apply to the same join point. The resulted behavior is similar to the application of multiple **around** advice pieces in AspectJ. Unfortunately, this behavior is not always desirable. For example, a reasonable composition of AspectJ and AspectWerkz might require that, at each join point, **before** advice, in both AspectJ and AspectWerkz aspects, are executed before any **around** advice, and finally followed by **after** advice. However, such an AspectJ/AspectWerkz composition is not achievable with wrapping semantics.

Our approach is not limited to the pipe-and-filter composition architecture. More complex composition semantics can be provided by imposing additional requirements on the aspect mechanism design. For example, one possibility is to specify types of aspectual effect that a mechanism can produce. With such a semantics, the overall aspectual effect can be constructed from aspectual effects of the collaborating mechanisms with regard to those effect types.

6.2 Other Mechanism Descriptions

Our choice of the mechanism's description style restricts access to the context data. Specifically, a mechanism can only access elements of the current or parent expression, environment, and stores. While this data can be sufficient for implementing the HisExt_1 and HerExt_2 aspect extensions for MyBase, real-world aspect extensions may generally require more information. For example, AspectJ needs access to callee and caller objects to construct a method call join point. Instantiating the approach for a description style that uses an explicit representation of the evaluation context (e.g., using a CEKS machine [14, 15]) would produce a more general solution.

In our solution we used annotated expressions to meet the granularity requirement of HisExt_1 and HerExt_2. The same result can be achieved by using small-step operational semantics for describing the mechanisms. In that case, each aspect mechanism would transform and extend certain base operational semantics rules.

7. RELATED WORK

7.1 Composing Aspect Extensions

Several authors point out the expressiveness drawback in using a single general-purpose AOP language, and emphasize the usefulness of combining modular domain-specific aspect extensions [12, 13, 21, 49, 39, 31]. However, the problem of composition has not received a thorough study.

7.1.1 XAspects

Shonle et al. [39] present a framework for aspect compilation that allows to combine multiple domain-specific aspect extensions. The framework's composition semantics is to reduce all extensions to a single general-purpose aspect extension (AspectJ). Specifically, given a set of programs written in different aspect extensions, XAspects produces a single program in AspectJ. An aspect extension program is translated to one or more AspectJ aspects. In XAspects,

collaboration between the aspect extensions is realized as a collaboration between the translated AspectJ's aspects.

The XAspects framework uses a translation-based approach. Specifically, XAspects translates programs in domain-specific aspect extensions to AspectJ. Unfortunately, in the presence of other aspects, this approach does not preserve the behavior of the domain-specific aspects, and therefore the XAspects approach does not guarantee a correct result.

Moreover, extensions in XAspects must be reducible to AspectJ. Since only a subset of aspect extensions is expressible in AspectJ, XAspects does not achieve composition in general. Our approach to composition and collaboration is not based on translation. In comparison to XAspects our approach is more general.

7.1.2 Concern Manipulation Environment

IBM's Concern Manipulation Environment provides developers with an extensible platform for concern separation: "The CME provides a common platform in which different AOSD tools can interoperate and integrate" [22]. CME would be a natural environment for a large scale application of our approach.

7.2 AOP Semantics

Existing works in AOP semantics explain existing aspect extensions and model AOP in general. We base some of our work on these studies. Unfortunately, they do not address the problem of aspect mechanism composition directly.

7.2.1 Semantics for Existing AOP Languages

Wand et al.'s [50] semantics for advice and dynamic join points explains a simplified dynamic AspectJ. They provide denotational semantics for a small procedural language, similar to ours. The language embodies key features of dynamic join points, pointcuts and advice. Their work, however, does not separate the AOP semantics from the base. Nevertheless, advice weaving is defined there as a procedure transformer. This is a special case of a language semantics transformer as we choose to define an aspect mechanism.

Method-Call Interception [28] is another semantical model that provides semantics for advising method calls. Similar to the previously discussed work, it highlights a very specific piece of AOP expressiveness (similar to AspectJ).

7.2.2 Semantical Models of AOP

Several studies of AOP semantics provide a general model of AOP functionality. Walker et al. [45] defines aspects through explicitly labeled program points and first-class dynamic advice. Jagadeesan et al. [24] uses similar abstractions (pointcuts and advice). Clifton et al. [8, 9] provides parameterized aspect calculus for modeling AOP semantics. In their model, AOP functionality can be applied to any reduction step in a base language semantics. This is similar to the definition of an aspect mechanism we use.

In comparison to our semantics, these models define AOP functionality using low-level language semantics abstractions. Using these more formal approaches for describing our method is left for future work.

7.2.3 Modular Semantics for AOP

We define an aspect mechanism separately from the base language and require it to specify only the AOP transformation functionality. This approach leads to the construction of modular AOP semantics. Exploring the application of other approaches for modular language semantics (e.g., modular SOS [32] and monad-based denotational semantics) to describing aspect mechanism is another area for further research.

8. CONCLUSION

In this paper, we address the open problem of integrating and using a base language Base with a set of third-party aspect extensions $\mathsf{Ext}_1, \ldots, \mathsf{Ext}_n$ for that language. We present a semantical framework in which independently developed, dynamic aspect mechanisms can be subject to third-party composition and work collaboratively.

We instantiate our approach for aspect mechanisms defined as expression evaluation transformers. The mechanisms can be composed like mixin layers [40, 35, 36] in a pipe-and-filter architecture with delegation semantics. Each mechanism collaborates by *delegating* or *exposing* the evaluation of expressions. The base mechanism serves as a terminator and does not delegate the evaluation further.

We applied our approach in the implementation of a concrete base language MyBase and two concrete aspect extensions to that language, HisExt_1 and HerExt_2. The implementation illustrates the construction steps. It provides semantics for third-party composition of aspect mechanisms.

The semantics for HisExt_1 resembles that for AspectJ. Indeed, our approach can be applied to implementing the pointcut and advice mechanism of AspectJ as an aspect extension to Java. This would provide a practical way to compose AspectJ with new domain-specific aspect extensions as they become available.

Acknowledgment

We thank Gene Cooperman, Darren Ng, Mitchell Wand, and the anonymous reviewers for their feedback and comments.

9. REFERENCES

[1] K. Arnold and J. Gosling. *The Java Programming Language*. The Java Series. Addison–Wesley Publishing Company, 1996.

[2] 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 Tarr [42], pages 87–98.

[3] D. Balzarotti and M. Monga. Using program slicing to analyze aspect-oriented composition. In C. Clifton, R. Lämmel, and G. T. Leavens, editors, *AOSD 2004 Workshop on Foundations of Aspect-Oriented Languages*, pages 25–29, Lancaster, UK, Mar. 23 2004. Technical Report 04-04, Department of Computer Science, Iowa State University Ames, Iowa, USA, Iowa State University.

[4] J. Bonér. What are the key issues for commercial AOP use: how does AspectWerkz address them? In *Proceedings of the 3rd International Conference on Aspect-Oriented Software Development*, pages 5–6, Mancaster, UK, 2004. AOSD 2004, ACM Press.

[5] J. Bonér. Invited talk: AspectWerkz 2 and the road to AspectJ 5. In *Invited Industry Talks at AOSD 2005*, Chicago, Illinois, USA, Mar. 14-18 2005. AOSD 2005.

[6] G. Bracha and W. Cook. Mixin-based inheritance. In *Proceedings of ECOOP/Object-Oriented Programming Systems, Languages, and Applications*, pages 303–311, Ottawa, Canada, Oct. 21-25 1990. OOPSLA'90, ACM SIGPLAN Notices 25(10) Oct. 1990.

[7] L. Cardelli, editor. *Proceedings of the 17th European Conference on Object-Oriented Programming*, number 2743 in Lecture Notes in Computer Science, Darmstadt, Germany, July21-25 2003. ECOOP 2003, Springer Verlag.

[8] C. Clifton, G. T. Leavens, and M. Wand. Formal definition of the parameterized aspect calculus. Technical Report TR #03-12, Dept. of Computer Science, Iowa State University, Oct. 2003.

[9] C. Clifton, G. T. Leavens, and M. Wand. Parameterized aspect calculus: A core calculus for the direct study of aspect-oriented languages. Technical Report TR #03-13, Dept. of Computer Science, Iowa State University, Nov. 2003.

[10] A. Colyer. AspectJ. In Filman et al. [16], pages 123–143.

[11] W. R. Cook. *A Denotational Semantics of Inheritance*. PhD thesis, Brown University, May 15 1989.

[12] C. Courbis and A. Finkelstein. Towards aspect weaving applications. In *Proceedings of the 27th International Conference on Software Engineering*, St. Louis, Missouri, USA, May 15-21 2005. ICSE 2005, ACM Press.

[13] K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 1st edition, 2000.

[14] M. Felleisen and D. Friedman. Control operators, the secd machine, and the lambda-calculus. *Formal Descriptions of Programming Concepts III*, pages 193–217, 1986.

[15] M. Felleisen and D. Friedman. A reduction semantics for imperative higher-order languages. In *Proceedings of the Parallel Architectures and Languages Europe 1987*, pages 206–223, 1987.

[16] R. E. Filman, T. Elrad, S. Clarke, and M. Akşit, editors. *Aspect-Oriented Software Development*. Addison-Wesley, Boston, 2005.

[17] P. Fradet and M. Südholt. AOP: towards a generic framework using program transformation and analysis. In S. Demeyer and J. Bosch, editors, *Object-Oriented Technology. ECOOP'98 Workshop Reader*, number 1543 in Lecture Notes in Computer Science, pages 394–397. Workshop Proceedings, Brussels, Belgium, Springer Verlag, July 20-24 1998. Extended version `http://www.irisa.fr/lande/fradet/PDFs/AOP98-long.pdf`.

[18] D. P. Friedman, M. Wand, and C. T. Haynes. *Essentials of Programming Languages*. MIT Press, Cambridge, MA, second edition, 2001.

[19] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Professional Computing. Addison-Wesley, 1995.

[20] S. Herrmann. Object teams: Improving modularity for crosscutting collaborations. In M. Akşit and M. Mezini, editors, *Proceedings of the 3rd International Conference Net.ObjectDays, NODe 2002*, Erfurt, Germany, Oct. 7-10 2002.

[21] J. Hugunin. The next steps for aspect-oriented programming languages (in Java). In *NSF Workshop on New Visions for Software Design & Productivity: Research & Applications*, Vanderbilt University, Nashville, TN, Dec. 13-14 2001. National Coordination Office for Information Technology Research and Development (NCO/IT R&D). White Paper.

[22] IBM's concern manipulation environment, 2004. `http://www.research.ibm.com/cme`.

[23] *Proceedings of the 7th ACM SIGPLAN International Conference on Functional Programming*, Uppsala, Sweden, Aug. 2003. ACM Press.

[24] R. Jagadeesan, A. Jeffrey, and J. Riely. An untyped calculus for aspect oriented programs. In Cardelli [7], pages 54–73.

[25] Jboss aspect oriented programming, 2005. `http://aop.jboss.org`.

[26] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In J. L. Knudsen, editor, *Proceedings of the 15th European Conference on Object-Oriented Programming*, number 2072 in Lecture Notes in Computer Science, pages 327–353, Budapest, Hungary, June 18-22 2001. ECOOP 2001, Springer Verlag.

[27] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Akşit and S. Matsuoka, editors, *Proceedings of the 11th European Conference on Object-Oriented Programming*, number 1241 in Lecture Notes in Computer Science, pages 220–242, Jyväskylä, Finland, June 9-13 1997. ECOOP'97, Springer Verlag.

[28] R. Lämmel. A semantical approach to method-call interception. In *Proceedings of the 1st International Conference on Aspect-Oriented Software Development*, pages 41–55, Enschede, The Netherlands, Apr. 2002. AOSD 2002, ACM Press.

[29] R. Lämmel. Adding Superimposition To a Language Semantics (Extended Abstract). In C. Clifton and G. T. Leavens, editors, *AOSD 2003 Workshop on Foundations of Aspect-Oriented Languages*, Boston, Massachusetts, Mar. 18 2003. Technical Report, Department of Computer Science, Iowa State University Ames, Iowa, USA, Iowa State University.

[30] C. V. Lopes. *D: A Language Framework for Distributed Programming*. PhD thesis, Northeastern University, 1997.

[31] C. V. Lopes, P. Dourish, D. H. Lorenz, and K. Lieberherr. Beyond AOP: Toward Naturalistic Programming. *ACM SIGPLAN Notices*, 38(12):34–43, Dec. 2003. OOPSLA'03 Special Track on Onward! Seeking New Paradigms & New Thinking.

[32] P. D. Mosses. Foundations of modular sos. In *MFCS '99: Proceedings of the 24th International Symposium on Mathematical Foundations of Computer Science*, pages 70–80. Springer-Verlag, 1999.

[33] G. C. Murphy, R. J. Walker, and E. L. A. Baniassad. Evaluating emerging software development technologies: Lessons learned from assessing aspect-oriented programming. *IEEE Transactions on Software Engineering*, 25(4):438–455, 1999.

[34] G. C. Murphy, R. J. Walker, E. L. A. Baniassad, M. P. Robillard, A. Lai, and M. A. Kersten. Does aspect-oriented programming work? *Comm. ACM*, 44(10):75–77, Oct. 2001.

[35] K. Ostermann. Implementing reusable collaborations with delegation layers. In D. H. Lorenz and V. C. Sreedhar, editors, *Proceedings of the First OOPSLA Workshop on Language Mechanisms for Programming Software Components*, pages 9–14, Tampa Bay, Florida, Oct. 15 2001. Technical Report NU-CCS-01-06, College of Computer Science, Northeastern University, Boston, MA 02115.

[36] K. Ostermann. Dynamically composable collaborations with delegation layers. In Cardelli [7], pages 89–110.

[37] J. A. Rees, W. D. Clinger, H. Abelson, N. I. Adams IV, D. H. Bartley, G. Brooks, R. K. Dybvig, D. P. Friedman, R. Halstead, C. Hanson, C. T. Haynes, E. Kohlbecker, D. Oxley, K. M. Pitman, G. J. Rozas, G. J. Sussman, and M. Wand. Revised^3 report on the algorithmic language Scheme. *SIGPLAN Notices*, 21(12):37–79, Dec. 1986.

[38] M. Shaw and D. Garlan. *Software Architecture, Perspectives on an Emerging Discipline*. Prentice-Hall, 1996.

[39] M. Shonle, K. Lieberherr, and A. Shah. XAspects: An extensible system for domain specific aspect languages. In *Companion to the 18th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 28–37, Anaheim, California, 2003. ACM Press.

[40] Y. Smaragdakis and D. Batory. Mixin layers: an object-oriented implementation technique for refinements and collaboration-based designs. *ACM Trans. Softw. Eng. Methodol.*, 11(2):215–255, 2002.

[41] C. Szyperski. *Component Software, Beyond Object-Oriented Programming*. Addison-Wesley, 2nd edition, 2002. With Dominik Gruntz and Stephan Murer.

[42] P. Tarr, editor. *Proceedings of the 4th International Conference on Aspect-Oriented Software Development*, Chicago, Illinois, USA, Mar. 14-18 2005. AOSD 2005, ACM Press.

[43] D. Thomas. Keynote: Transitioning AOSD from research park to main street. In Tarr [42], page 2.

[44] W. Vanderperren, D. Suvée, B. Verheecke, M. A. Cibrán, and V. Jonckers. Adaptive programming in JAsCo. In Tarr [42].

[45] D. Walker, S. Zdancewic, and J. Ligatti. A theory of aspects. In ICFP 2003 [23], pages 127–139.

[46] R. J. Walker, E. L. A. Baniassad, and G. Murphy. Assessing aspect-oriented programming and design. In C. Lopes, G. Kiczales, B. Tekinerdoğan, W. De Meuter, and M. Meijers, editors, *Workshop on Aspect Oriented Programming (ECOOP 1998)*, June 1998.

[47] R. J. Walker, E. L. A. Baniassad, and G. C. Murphy. An initial assessment of aspect-oriented programming. In *Proc. 21st Int'l Conf. Software Engineering (ICSE '99)*, pages 120–130, 1999.

[48] R. J. Walker, E. L. A. Baniassad, and G. C. Murphy. An initial assessment of aspect-oriented programming. In Filman et al. [16], pages 531–556.

[49] M. Wand. Understanding aspects (extended abstract). In ICFP 2003 [23]. Invited talk.

[50] M. Wand, G. Kiczales, and C. Dutchyn. A semantics for advice and dynamic join points in aspect-oriented programming. *ACM Trans. Prog. Lang. Syst.*, 26(5):890—910, Sept. 2004.

[51] J. C. Wichman. ComposeJ: The development of a preprocessor to facilitate composition filters in the Java language. Master's thesis, Department of Computer Science, University of Twente, Enschede, the Netherlands, Dec. 1999.