Reflective Mechanisms in AOP Languages

Sergei Kojarski David H. Lorenz*
Northeastern University
College of Computer & Information Science
Boston, Massachusetts 02115 USA
{kojarski,lorenz}@ccs.neu.edu

ABSTRACT

The paper describes concepts and experiments with reflective mechanisms in AOP languages, illustrating that an AOP computation is a reflective computation. The goal is to improve the understanding of AOP and reflection by revealing that AOP and reflection are in fact both first class reflective mechanisms. The paper discusses the relation between reflection and AOP, and demonstrates how AOP can be used for implementing reflection. We illustrate that in case the aspect language does not offer core reflection or in case the reflection offered by the aspect language is not expressive enough, reflection can be simulated with AOP. This is in contrast to the traditional thinking of AOP as merely a mechanism for facilitating reflection functionality.

1. INTRODUCTION

Aspect-oriented programming (AOP) is in essence a computational reflection mechanism [17]. The join point model reflects a program's behavior: a join point provides the ability to introspect; advice provides the intercession (manipulation) capability. Aspect-oriented languages (e.g., AspectS [7] and AspectJ [15, 10]) are typically built on top of a base object-oriented programming (OOP) language, which has native support for reflection. Consequently, in addition to its aspectual reflective mechanism ($\mathcal{AM}$), the aspect-oriented language also supports the underlying OOP core reflection mechanism ($\mathcal{RM}$). AOP programmers thus have two reflective mechanisms to their disposal.

From a software engineering point of view, the two reflective mechanisms serve different purposes, with different application areas. In AspectJ, for example, Java Core Reflection is mainly used for introspection on structure [4], while aspects are mainly used for intercession on behavior. However, an aspect could just introspect the behavior without inflecting any functional affect on the program; and instantiating objects or invoking methods using reflection does affect the program's run-time structure and behavior.

*Supported in part by the National Science Foundation (NSF) under Grant No. CCR-0098643 and CCR-0204432, and by the Institute for Complex Scientific Software at Northeastern University.

[0]A short version of this paper appeard as a position paper at AOSD'03 SPLAT Workshop [11].

The goal of the paper is to improve our understanding of AOP and reflection by revealing that AOP and reflection are in fact both first class reflective mechanisms. This paper explores to what extend reflective properties of $\mathcal{AM}$ and $\mathcal{RM}$ overlap. We illustrate how $\mathcal{AM}$ and $\mathcal{RM}$ may interact in beneficial and unexpected ways, resulting in improvement of the software engineering abilities of each other. The paper addresses the fundamental question of whether or not one mechanism subsumes the other.

We show that the two mechanisms interact, and making practical observations based on those examples. We illustrate that traditional AOP semantics can be achieved by reflection; we illustrate that traditional reflection semantics can be simulated by AOP; and we give examples on how the two can interact collaboratively.

The rest of the paper is organized as follows. Section 2 illustrates the ability of AOP to simulate reflection by giving two concrete examples in Java and AspectJ. Section 3 provides the conceptual framework to viewing AOP as computational reflection. Section 4 describes experiments in implementing a complete reflection application programming interface (reflection API) on top of AspectJ without using Java Core Reflection. In Section 5, the *in*ability of AOP to do every single thing that core reflection supports is exposed clearly, and similarly in the opposite direction. We also describe a practical software engineering perspective on AOP and reflection collaboration. Related work is described in Section 6, and in Section 7 we conclude by mapping the space of AOSD languages with respect to aspectual and reflective features.

2. MOTIVATION

Historically, the seeds of AOP are founded in metaobject protocols (MOPs) [9] and open implementations (OIs) [8]. This fact gave rise to the point of view that reflection is the underlying basis for AOP. Under this view [18], AOP facilitates disciplined metaprogramming, utilizing MOPs [9]. It is sometimes even said that the emerging aspect languages reify principled reflection "meta-level" abilities to "base-level" programmers.

Indeed, AOP can be implemented as an aspectual interface (AI) to the underlying (MOP) reflective interface (RI), as exemplified by AspectS. However, an AOP system is reflective in its own right. Conceptually, the AOP language can implement its AI directly. For example, MCI [12] is a statically type-safe extension of an OOP language, μO^2 [12], with native support for AI. While a compiler for MCI hasn't been implement yet, MCI has complete semantics and no reflection.

Moreover, in this paper we show that AI provides almost the same level of introspection as RI. We show that AOP offers meta-information at a comparable level to that of full introspection MOP's, (i.e., we show that it is not true that join point models offer less meta-information than introspection.)

Throughout the paper we use RI and AI as abbreviations for the interfaces provided by reflective and aspectual features, respectively. We differentiate between $\mathcal{AM}$- and $\mathcal{RM}$-based implementation of these feature interfaces. We denote by $\mathrm{RI}^{\mathcal{R}}$ and $\mathrm{AI}^{\mathcal{R}}$ the $\mathcal{RM}$-based implementation, and by $\mathrm{RI}^{\mathcal{A}}$ and $\mathrm{AI}^{\mathcal{A}}$ the $\mathcal{AM}$-based implementation, of RI and AI, respectively.

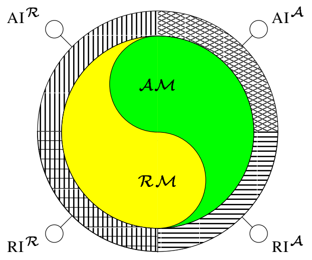
$\mathrm{AI}^{\mathcal{R}}$
$\mathrm{AI}^{\mathcal{A}}$
$\mathcal{AM}$
$\mathcal{RM}$
$\mathrm{RI}^{\mathcal{R}}$
$\mathrm{RI}^{\mathcal{A}}$

Figure 1: Aspectual Reflection

In the rest of this section, we underscore the observation that AI can be the basis for RI. Even if the base OO language does not provide $\mathrm{RI}^{\mathcal{R}}$, the AOP language can still build one $\mathrm{RI}^{\mathcal{A}}$ on top of its $\mathcal{AM}$. We illustrate this in Java and AspectJ first for object-level and then for class-level structural reflection.

2.1 AOP-based Object-level Reflection

Consider the manner in which a client of reflection can access the field `fname` of an object `host` using Java Core Reflection:

```
host.getClass().getDeclaredField(fname).get(host)
    ;
```

which actually should be in a **`try-catch`** statement:

```
try {
  host.getClass()
    .getDeclaredField(fname).get(host);
} catch(IllegalAccessException iae) {
  System.out.println("Illegal field access: "
    + iae.getMessage());
} catch(NoSuchFieldException nsfe) {
  System.out.println("Field not found: "
    + nsfe.getMessage());
}
```

Now assume AspectJ was an extension of a dialect of Java that did not include support for core reflection. We could still provide access field values using AOP *without* relying on reflection by, e.g.,

```
FieldInspector.aspectOf(host).get(fname);
```

where reflection is simulated with the `FieldInspector` aspect:

```
import java.util.HashMap;
aspect FieldInspector
 pertarget(initialization(
    (!FieldInspector).new(..))) {
 before(Object newValue): set(* *.*)
   && args(newValue) {
   fields.put(thisJoinPoint
     .getSignature().getName(),newValue);
 }
 public Object get(String name) {
   return fields.get(name);
 }
 private HashMap fields = new HashMap();
}
```

In the above aspect, **`pertarget`** guarantees that every object ever created is associated on initialization with an instance of the `FieldInspector` aspect. `FieldInspector.hasAspect(`**`target`**`)` is always true, and `aspectOf(host)` is always defined. For simplicity, `FieldInspector` assumes field names to be unique (and ignores the issue of overridden fields) and (similar to $\mathrm{RI}^{\mathcal{R}}$) boxes primitive values in objects (ignoring default value initialization).

The `FieldInspector` example illustrates that it is possible to provide reflective information about every field of every object without having core reflection. This example can be extended to provide complete structural reflection on the object-level resulting in the AOP-based RI ($\mathrm{RI}^{\mathcal{A}}$). $\mathrm{RI}^{\mathcal{A}}$ could even expose structural information beyond what $\mathrm{RI}^{\mathcal{R}}$ normally offers. For example, run-time profiling information (e.g., how many times a particular method was executed) is not a part of $\mathrm{RI}^{\mathcal{R}}$, but can be easily provided using AOP.

2.2 AOP-based Class-level Reflection

To complete the picture, we need to also explore reflection over a class structures. Consider the manner in which one can find a superclass of a class `C` using Java Core Reflection API ($\mathrm{RI}^{\mathcal{R}}$):

```
C.class.getSuperclass();
```

Similar to the case with object-level reflection, aspects can also simulate class-level reflection. We can find the superclass using AOP *without* using $\mathrm{RI}^{\mathcal{R}}$ by, e.g.,

```
ClassHierarchy.getSuperclass(C.class)
```

where the class-level structural reflection is simulated with the aspect `ClassHierarchy`:

```
import java.util.HashMap;
aspect ClassHierarchy
 percflow(call(
    (!ClassHierarchy).new(..))) {
 before(): initialization(
    (!ClassHierarchy).new(..)) {
   Class descClass = thisJoinPointStaticPart
     .getSignature().getDeclaringType();
   hierarchy.put(descClass,superClass);
   superClass = descClass;
 }
 private static HashMap hierarchy =
   new HashMap();
 public static Class getSuperclass(
   Class desc) {
   return (Class)hierarchy.get(desc);
 }
 private Class superClass;
}
```

The `ClassHierarchy` aspect maintains a hash-map `hierarchy`, which reflects all the classes instantiated directly or indirectly.

3. CONCEPTS

This section provides the conceptual framework for discussing the relation between reflection and AOP. The goal is to remove the haze from the understanding of AOP and reflection by revealing that AOP and reflection are in fact (almost) equivalent concepts. Traditionally, AOP is viewed as a reflection-based or transformation-based extension to a certain core OOP language. Conceptually, however, reflection can also be viewed as an AOP-based extension to a certain core AOP language.

3.1 Computational reflection

A system exhibits computational reflection [17] if:

(i) the system always has an accurate self-representation; and

(ii) the system is causally connected to its self-representation.

Figure 2 depicts the similarities between reflection and AOP as first-class computational reflection mechanisms. The circle-head lines marked $\text{RI}^{\mathcal{R}}$ and $\text{AI}^{\mathcal{A}}$ are the interfaces used by clients of reflection and AOP. $\mathcal{D}^{\mathcal{R}}$ and $\mathcal{D}^{\mathcal{A}}$ are self-representation domains of meta-information. The internal information and the domains $\mathcal{D}^{\mathcal{R}}$ and $\mathcal{D}^{\mathcal{A}}$ are linked in such a way that if one of them changes, the other changes correspondingly. The arrows $\mathcal{RM}$ and $\mathcal{AM}$ are the mechanisms that keep the system's internal information causally connected to $\mathcal{D}^{\mathcal{R}}$ and $\mathcal{D}^{\mathcal{A}}$, respectively. Some of the internal information is reflected by both $\mathcal{D}^{\mathcal{R}}$ and by $\mathcal{D}^{\mathcal{A}}$, but they do not overlap entirely.

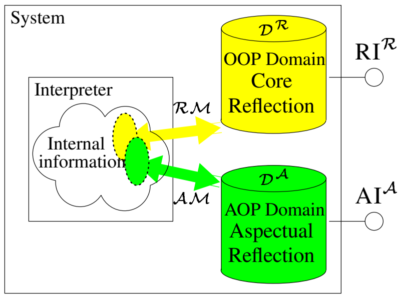
System
Interpreter
Internal information
$\mathcal{RM}$
$\mathcal{AM}$
$\mathcal{D}^{\mathcal{R}}$
OOP Domain Core Reflection
$\mathcal{D}^{\mathcal{A}}$
AOP Domain Aspectual Reflection
$\text{RI}^{\mathcal{R}}$
$\text{AI}^{\mathcal{A}}$

Figure 2: Native $\mathcal{AM}$ and native $\mathcal{RM}$

3.2 Core Reflection

Core reflection comprises a reflection API, $\text{RI}^{\mathcal{R}}$, an OOP domain meta-information, $\mathcal{D}^{\mathcal{R}}$, and a reflective mechanisms, $\mathcal{RM}$, keeping the system causally connected to the meta-information.

For example, in Java, $\text{RI}^{\mathcal{R}}$ is the package `java.lang.reflect`. $\mathcal{D}^{\mathcal{R}}$ is the OOP reflective meta-information domain, e.g., Java Core Reflection, comprising of the class objects, method objects, etc. that reify $\text{RI}^{\mathcal{R}}$. $\mathcal{RM}$ denotes reflective mechanism that keeps the system's internal state causally connected to $\mathcal{D}^{\mathcal{R}}$. A change to the internal information is reflected in $\mathcal{D}^{\mathcal{R}}$ and a change in $\mathcal{D}^{\mathcal{R}}$ is reflected internally, but in Java, the structural information in $\mathcal{D}^{\mathcal{R}}$ is read-only.

3.3 Aspectual Reflection

An AOP system comprises AI, $\mathcal{D}^{\mathcal{A}}$, and $\mathcal{AM}$. Consider an AOP language with native supports for AI. We denote by AI the aspectual interface to the aspectual domain $\mathcal{D}^{\mathcal{A}}$ (i.e., the syntax of a certain AOP expressiveness); and by $\mathcal{AM}$ the aspectual mechanism that keeps the internal state casually connected to $\mathcal{D}^{\mathcal{A}}$ (i.e., the mechanism that preserves the semantics of AI).

Next we elaborate on what we mean by AI, $\mathcal{D}^{\mathcal{A}}$, and $\mathcal{AM}$. The rest of this section is leading to the logical conclusion that AOL supports "core" computational reflection via AI, just as OOL support computational reflection via RI, and thus AOP languages have a reflective architecture.

3.4 What is AI?

An AOP system is a system with some implementation of AI. Generally, an AOP language comprises a base language and an aspectual interface AI. The dynamic semantics of AI denotes aspectual computations.

3.5 What is $\mathcal{D}^{\mathcal{A}}$?

An aspectual computational system is a system whose purpose is to support action in the AOP domain, where the AOP domain is the domain of the base language plus the aspectual domain. For this paper we only assume that there is some definition of a join point model (JPM), some definition of an advice model (ADM), and some definition of a superimpose model (SIM), where SIM is a relation over JPM $\times$ ADM. The aspectual domain consists of JPM, ADM, and a relation SIM. That is, $\mathcal{D}^{\mathcal{A}}$comprises a join point model, an advice model, and a superimpose model.

JPM specifies the types of join points and their structure. ADM specifies the kinds of advice and how they modify the program behavior during execution. SIM is a mapping JPM $\rightarrow$ ADM, which prescribes which advice applies to which join point.

SIM can be implicit or explicit. In AspectJ, for example, the superimpose instruction is implicit by including, within the advice, point cut descriptors (PCD), i.e., a patterns specification of sets of join points. In MCI, on the other hand, the superimpose instruction is explicit using a `superimpose` language construct that allows to attach an advice to join points of a specified method calls.

3.6 Self-representation

In order to justify that the internal information is causally connected to $\mathcal{D}^{\mathcal{A}}$, we must show that $\mathcal{D}^{\mathcal{A}}$ is indeed a self-representation of the system. A system's self-representation means some abstraction of itself, which must be *accurate but not necessarily complete*. That is, some image of the internal-information is causally connect to the system's meta-information, namely its self-representation.

In AOP, a JP is a point in the program execution: it is an abstraction of a state transitions in the program execution. The join point model specifies what are the kinds of monitored transitions (i.e., kinds of join points). Monitored state transitions in the execution lead to corresponding join points. The join point model is linked to the program execution is such a way that it reflects the execution.

Join points reflect the transitions; advice affects the transitions, which are then reflected in the join points. Join points are temporally connected to internal behavior information (state transition.) We can say that the two are causally connected.

Hence, $\mathcal{D}^{\mathcal{A}}$ is a self-representation of the program execution, because

(a) monitored state transitions are accurately represented by the join points; and

(b) advising join points affects the internal state transitions.

3.7 What is $\mathcal{AM}$?

$\mathcal{AM}$ is the implementation of the AI semantics. $\mathcal{AM}$ is the mechanism that guarantees that the internal information and the external self-representation are causally connected. Internal means program transition between states. External means the join point model. The execution is a dynamic representation of the program. $\mathcal{AM}$ is a temporal state of the program in the form of a steam of join points. The stream of join points reflects the execution of the program; and the join point stream is in sync with the execution. $\mathcal{AM}$ clients (e.g., advice) affect the execution of the program. As a result, $\mathcal{AM}$ clients change the stream of join points.

Hence aspectual computation is a reflective computation, i.e.,:

COROLLARY 1. *An AOP system is reflective.*

3.8 Aspectual Architectures

Now that we have established conceptually that an AOP system is a reflective system, we list three different aspectual architectures exhibiting different implementation strategies for AI :

Native. AI is the aspectual instruction set that clients use for AOP. The actual processor that interprets AI instructions can be a native $\mathcal{AM}$, as shown in Figure 2 (but without the reflection parts $\mathcal{RM}$,$\mathcal{D}^{\mathcal{R}}$, and RI$^{\mathcal{R}}$.) MCI is an example of an AOP language, which is based on a pure OOP language (supports class hierarchies with single inheritance). The base OOP language does not provide support for reflection. MCI is a language extension that is type-safe, supports separate compilation, it is state-aware (evaluation of advice code could produce side-effects). MCI provides precise operational semantics for a native implementation of AI: the precise semantics for the superimpose construct is defined directly. While a compiler has not been implemented for the MCI language, such a compiler is implementable.

Reflection-based AI may be implemented over an underlying $\mathcal{RM}$, as is the case for AspectS. The AOP instructions include AsBeforeAfterAdvice etc. The implementation is in Smalltalk, that is, the processor is Smalltalk. The operational semantics is given in terms of method wrappers [3].

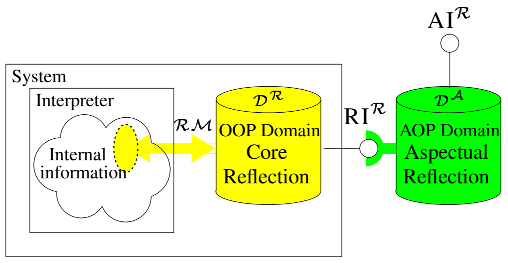
AI$^{\mathcal{R}}$
System
Interpreter
Internal information
$\mathcal{RM}$
$\mathcal{D}^{\mathcal{R}}$
OOP Domain Core Reflection
RI$^{\mathcal{R}}$
$\mathcal{D}^{\mathcal{A}}$
AOP Domain Aspectual Reflection

Figure 3: Reflection-based $\mathcal{AM}$

Transformation-based DemeterJ is an AOSD system that implement by transformation. AOP instructions, such as Before etc., are implement by transforming to Java operational semantics. We denote by AI$^{\mathcal{T}}$ the aspectual interface to a transformation-based $\mathcal{AM}$.

AspectJ is an example of partly reflection-based and partly transformation based as shown in Figure 4. The semantic of AspectJ was initially "transformation-based" (static + dynamic JPM). Later AspectJ adapted its own native JPM, although the implementation remained transformation-based and reflection-based, using the Java virtual machine (JVM). Future versions of AspectJ might eventually include a JVM with native support for $\mathcal{AM}$. A JVM with a native $\mathcal{AM}$ would allows, e.g., "aspectof" in addition to "instanceof".

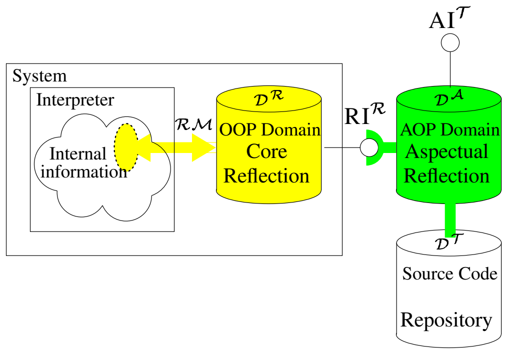
AI$^{\mathcal{T}}$
System
Interpreter
Internal information
$\mathcal{RM}$
$\mathcal{D}^{\mathcal{R}}$
OOP Domain Core Reflection
RI$^{\mathcal{R}}$
$\mathcal{D}^{\mathcal{A}}$
AOP Domain Aspectual Reflection
$\mathcal{D}^{\mathcal{T}}$
Source Code Repository

Figure 4: Transformation-based $\mathcal{AM}$

3.9 Reflective Architectures

There are several possible implementation strategies for RI too. RI is the reflective instruction set that clients use for OOP reflection.

Native. The actual processor that interprets RI instructions can be natively supported, i.e., core reflection $\mathcal{RM}$. This is the case in Java Core Reflection as well as in Smalltalk's MOP.

Repository-based. Elsewhere [16], we illustrate that the reflective meta-interface can be decoupled from its reflection implementation, and that clients can be retargeted to use a mirrored reflection instead. We call that *pluggable reflection* [16].

AOP-based. Last but not least, the reflective mechanism could be simulated by an aspectual mechanism. That is, imagine AspectJ had core $\mathcal{AM}$ but no core $\mathcal{RM}$, we could still simulate RI$^{\mathcal{A}}$ on top of $\mathcal{AM}$.

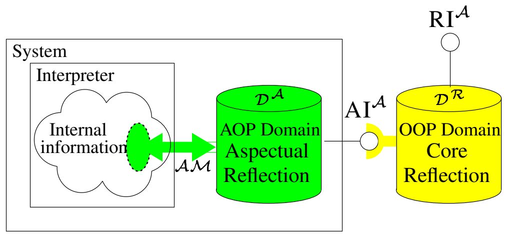
RI$^{\mathcal{A}}$
System
Interpreter
Internal information
$\mathcal{AM}$
$\mathcal{D}^{\mathcal{A}}$
AOP Domain Aspectual Reflection
AI$^{\mathcal{A}}$
$\mathcal{D}^{\mathcal{R}}$
OOP Domain Core Reflection

Figure 5: Aspect-based $\mathcal{RM}$

In the next section we illustrate concretely that AOP-based reflection is feasible in AspectJ.

4. EXPERIMENTS

This section descibes an experiment in aspectual reflection striving to verify the following hypothesis:

HYPOTHESIS 1. *AspectJ's $\mathcal{AM}$ subsumes Java's $\mathcal{RM}$.*

In order to assess to what degree Hypothesis 1 holds, we build an AOP-based (aspectual) reflection in AspectJ. Specifically, we implement Java Core Reflection API using aspects and illustrate that clients of Reflection API can be retargeted to use aspectual reflection instead of Core Reflection. The goal is to demonstrate that AspectJ's aspectual domain is able to provide as much meta information as Java Core Reflection does.

4.1 Requirements

It is important to clarify what information is allowed to be accessed from within the implementation of aspectual reflection, and what outcome would count as a successful result of the experiment.

Generally, the implementation of aspectual reflection is allowed to only use DA as a source of meta-information without accessing DR. It means that reflective data is obtained via join points along (reflected) program execution while functionality provided by `java.lang.Class`, `java.lang.reflect.*` and native runtime type information (RTTI) (`java.lang.Object.getClass()`) is forbidden.

On practice, however, we allow calls to `Class.toString()` and `Class.getModifiers()` methods. This exception is caused by the fact that AspectJ doesn't provide it's own abstraction over types and uses `java.lang.Class` to expose join point type information (e.g. org.aspectj.lang.Signature, org.aspectj.lang.reflect.SourceLocation). We regard these methods to be pieces of information that conceptually could be exposed via the join point interface in reflection - independent manner (for example, modifiers for a particular class could be exposed via `staticinitialization` join point signature while string representation of types could be a part of any join point signature).

To avoid any misunderstanding and to check that no illegal calls are made we use the `Proof` aspect shown at Listing 1. The aspect statically verifies via the **`declare warning`** construct of AspectJ, that no calls are made to Java Core Reflection API from the aspectual reflection implementation. Its sole purpose is to be a proof that we are not "cheating".

Listing 1: Proof.java

```
aspect Proof {
 declare warning: call(* java.lang.Class.*(..))
   && !call(* java.lang.Class.getName(..))
   && !call(* java.lang.Class.getModifiers(..))
   && !call(* java.lang.Class.isInterface(..)):
   "Using Java Core Reflection!";
}
```

4.2 Methodology

Two general issues need to be adressed by the implementation:

1. Obtaining reflective information from a program execution;
2. Retargeting client code to use aspectual reflection instead of Core Reflection.

We use an aspect-oriented method to deal with both concerns by associating specific group of aspects with each.

Collector Several `Collector` aspects employ $\mathcal{AM}$ to collect meta-information from a reflected program by advising join points along its execution path.

Retarget A `Retaget` aspect replaces functionality provided by Core Reflection with aspectual reflection mechanism.

4.3 Model

The overall architecture of the solution is shown in Figure 6. The

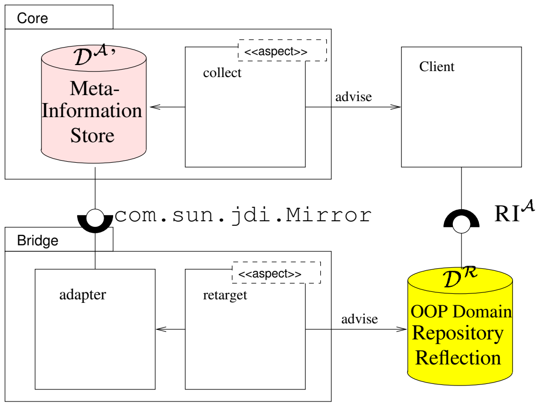

Figure 6: Overall architecture

model consists of the following two packages:

- Core. Core package is the heart of the system. It addresses issues of obtaining and storing meta information about reflected program . The package consists of the following modules:
  - *Collector aspects* advise reflected program. Provided abstraction of a program execution as join point sequence, aspects access meta-information via join point interface and place it into the aspectual repository DA;
  - *Aspectual repository* is a collection of classes that implements interface, similar to (but simpler than) `com.sun.jdi.Mirror`. In essence, it is a database of meta-information. The repository implements two interfaces:
    - ∗ Repository - Collector interface. The interface allows `Collector` aspects to deposit meta-information into the database and is internal for the package (package - protected);
    - ∗ `com.sun.jdi.Mirror` interface allows external clients to access content of the repository. The interface supports clients with both static (types) and (dynamic) meta-information about reflected program.
- Retarget. Retarget package takes care of redirecting reflection client to use aspectual repository and performs all the necessary conversions between Reflection API and aspectual repository terms. It consists of the two modules:

- *Retarget* contains set of **around** advice wrapping executions of Core Reflection API methods so that original methods never executes but aspectual repository is used instead. All the calls to aspectual repository from `Retarget` are done indirectly via `Adapter` module;
- *Adapter* is a bridge performing conversion between Reflection API and repository terms. It is implemented as a Reflection API - compatible wrapper around repository classes.

Finally, client's code still uses RI (the reflection interface) but is actually served with information from $\mathcal{D}^{\mathcal{A}}$ filled by $\mathcal{AM}$ thereby using RI$^{\mathcal{A}}$.

4.4 Implementation

This section provides an implementation of the system model. Following the model, full code is given in 4 steps:

1. Collector aspects;
2. Aspectual repository;
3. Retarget aspect;
4. Adapter classes;

4.4.1 Collector Aspects

The `Collector` aspects advise client's code in order to intercept join points that occur during a client program execution. There are two collector aspects:

- A `EntityCollector` keeps the aspectual repository in sync with the object store of the interpreter by updating it with newly-created instances. It also obtains structural information over class hierarchies of a reflected program. Finally, it monitors if constructed object is of inner non-static class and, if so, creates inner/enclosing link between created and creator objects in the object store;
- A `MemberCollector` supplies the repository with information about class and interface members, such as fields (and their values), methods, constructors and inner classes.

Listing 2: EntityCollector.java

```
package edu.neu.ccs.mirror.aspects.internal;
import java.util.Vector;

import org.aspectj.lang.reflect.*;

aspect EntityCollector dominates MemberCollector
  percflow(Any.ConstructorCall()) {
 private Object created=null;
 private CodeSignature sig;
 private Vector interfaces=new Vector();
 private Vector classes=new Vector();
 private ClassType superclass;

 /**
  * Runs once per each aspect instance
  * That's how we know static type of
  * object is about to be created.
  */
 before():Any.ConstructorCall() {
  sig = (CodeSignature)thisJoinPointStaticPart.
       getSignature();
 }

 /**
  * At first initialization jp within constructor call flow
  * adds new object into the object store.
  * For each new initialization join point
  * builds correspnding links for a class
  * hierarchy
  */
 before(Object created):Any.Initialization(created
      ) {
  //Adding new object into the object store
  if (this.created==null) {
   this.created=created;
   //Add object into value store
   //Note, that information about object type
   //is known from constructor signature
   Store.addObject(created,sig);
  }

  //Changing superclass information for types in the
  //type store
  Method method = Store.convert((CodeSignature)
                  thisJoinPointStaticPart.
                       getSignature());
  ReferenceType declaringType = method.
       getDeclaringType();

  if (declaringType instanceof ClassType) {
    ClassType subclass = (ClassType)declaringType;
    //if (!subclass.isSuperAssigned())
    subclass.setSuperClass(superclass);
    superclass = subclass;
    //at this point interfaces vector contains
    //all the interfaces this class implements
    //(directly and by extending its superclasses)
    //Of course, implementation is not efficient...

    for (int i=0;i<interfaces.size();i++) {
     InterfaceType iType= (InterfaceType)interfaces
          .get(i);

     //1. Adding current class as implementor
     // to the interface
     iType.addImplementor(subclass);

     //2. Adding the interface to
     //the vector of interfaces implemented
     //by the current class
     subclass.addInterface(iType);
     //System.out.println("adding interface "+iType.getName()+" to
          class "+subclass.getName());
    }
  }
  if (declaringType instanceof InterfaceType)
   interfaces.add(declaringType);
 }
}
```

Adding object. `EntityCollector` (Listing 2) aspect advises object-creation join points of the client program. Using capabilities of context exposure provided by AspectJ, it accesses newly created objects and places them into the repository.

Notice that object are added by the static method of `Store.addObject` , providing it the `sig` actual argument along with the object itself. `sig` carries information about the type of the object created. `sig` is obtained by an advice on the `Any.ConstructorCall` pointcut that captures all the constructor calls occurring within the client program. Observe that a constructor call join point always precedes the corresponding object initialization join point, thus allowing us to safely use this technique.

Furthermore, the object creation can trigger more than one initialization join point. More specifically, there are initialization join points for each interface that the object's class implements and for each superclass of the object's class. The private variable `created` defined in the aspect guards the repository from excessive updates, so that an object is added only once at the first initialization join point. Also note that the **`before`** construct ensures that the object will be accessible via the aspectual repository as soon as it is created (and even before it is used by the base program).

Reconstructing the class hierarchy. As mentioned above, an initialization join point is triggered not only for a class instantiating the object, but also for each interface this class implements as well as for each superclass in the hierarchy.

Furthermore, AspectJ defines the order of these join points as follows:

1. All the classes are traversed from superclass to subclass order, starting from the the root of hierarchy;
2. In each level, first the initialization join points for the interfaces that the class implements are triggered. Then, class-level initialization is done with the current-level initialization join point. "Current-level initialization join point" and "interface initialization join point" means that the signature of the join point contains, at the declaring class, the class of the current level or the interface.

By relying on this protocol and the predicted order of join points, `EntityCollector` can easily recognize the class hierarchy for the type of the object being constructed.

All the static information is obtained via **`thisJoinPointStaticPart`** of a join point. Conversion from AspectJ signature class instances to the repository `Method` class is implemented in the `Store` class.

As can be seen, `declaringType` of the initialization join point could be of either InterfaceType or ClassType. If it is InterfaceType, we simply add it into the interfaces vector. If it is a ClassType, this means that the current initialization join point completed the current class level in the initialization path. Thereby, we add into the store the information about this level that includes the interfaces and superclass (hierarchy) information.

In fact, `EntityCollector` sees each constructor call as separate unit to be analyzed. To reflect this fact, the `EntityCollector` aspect is defined as **`percflow`**`(Any.ConstructorCall)`. This guarantees that:

- The existence of one instance of the aspect per each constructor call. Therefore, the aspect's fields will only be relevant to the particular object construction.
- If any of the object constructors calls constructors on program classes within it's code, then for that call a new `EntityCollector` will be instantiated, without interfering with the current instance of `EntityCollector`.

Recognizing enclosing/inner object relationship. Enclosing/inner object relationship can be easily recognized, since:

- Inner object is always instance of non-static inner class;
- Inner object could only be constructed by enclosing object (in AspectJ's terms, for inner object constructor call join point, "currently executing" object is always enclosing one).

In the `EntityCollector` aspect, private field `creator` always contains "currently executing" object of the monitored constructor call, as guaranteed by the body of **`before`**`():Any.ConstructorCall()` advice.

As the first initializer for a constructed object runs, aspect checks if `created` object type is inner and non-static. If so, `created` object is inner object of `creator` instance.

Listing 3: MemberCollector.java

```
package edu.neu.ccs.mirror.aspects.internal;
import org.aspectj.lang.*;
import org.aspectj.lang.reflect.*;

aspect MemberCollector {
 /**
   * Adds new types into the store.
   */
 before(): Any.StaticInitialization() {
  Store.convert(thisJoinPoint.getSignature().
       getDeclaringType());
 }

 /**
  * Field type and code tables
  * meta-data collector via
  * get join point.
  * (programs can access fields
  * not covered by set join point, i.e.
  * not-initialized (initialized with
  * default value) fields.
  */
 after() returning(Object value):
               Any.Get() {
   Store.updateField(thisJoinPoint.getTarget(),
        value,
                  (FieldSignature)
                       thisJoinPointStaticPart.
                  getSignature());
 }

 /**
  * Field type and code tables
  * meta-data collector via
  * set join point.
  */
 before(Object value):
      Any.Set() && args(value) {
   Store.updateField(thisJoinPoint.getTarget(),
        value,
            (FieldSignature)thisJoinPointStaticPart
                  .
            getSignature());
 }

 /** Method type & code tables
  * meta-data collector
  * for methods only
  */
 before():Any.MethodCall() || Any.MethodExecution
      () {
```

```
  Store.updateMethod(thisJoinPoint.getTarget(),
      null,
             (CodeSignature)thisJoinPointStaticPart
                 .
              getSignature());
 }

/** Method type & code tables
  * meta−data collector
  * for constructors only
  */
before():Any.ConstructorExecution() {
  Store.updateMethod(null,null,
             (CodeSignature)thisJoinPointStaticPart
                 .
              getSignature());
 }
}
```

The `MemberCollector` aspect is a singleton aspect that advises all the member-relevant join points in the program flow. All the object state information (field values) is obtained via the join point interface, while all the information related to the member types is obtained via the join point signatures. MemberCollector also advises all the program `staticinitialization` join points thereby adding class and interface types into the store as soon as JVM runs their static initializer.

Reflecting on members *Field* types and field values are gathered by advising field accesses and settings throughout the program (`Any.Get()` and `Any.Set()`) Conceptually, the **set** join point would be enough to cover all field-related information. However, AspectJ doesn't intercept default field settings. If it were the case, `Any.Get()` advice would be excessive.

Method types are gathered by advising all method calls and executions of the base program. Interesting to note that the signature of a method call is not always the same as a method execution signature due to dynamic method dispatch mechanism. For example, abstract methods meta-information is never revealed via method execution, but only through method call join points.

Constructors are never abstract. Therefore, advice on constructor execution join points is sufficient.

Adding class and interface types Shortly after loading class, JVM runs its static initializer. Aspect intercepts those executions by **before** advice to `Any.StaticInitialization()` pointcut. Unlike well-defined protocol for object initialization, JVM doesn't provide sufficent logic for a sequence of class-loads that would allow to construct class and interface hierarchies at load-time. Therefore, advising `staticinitialization` join points doesn't do much but only places new "stub" type into the type store (type without any information about it's type hierarchy). However, knowledge of a type name allows to reveal if the type is inner class. It is plausible, because:

- for all inner classes, name of inner class includes name of all of its enclosing classes;
- for non-static inner classes, immediate enclosing class is always loaded and initialized before inner class;
- for static inner classes, attempt to load inner class triggers load of its top-most enclosing class;

Thanks to this logic, we are able to collect inner-type hierarchy meta information.

Any aspect All the pointcuts used in the collector aspects are are defined inn the `Any` aspects. Code of the `Any` aspect is shown at Listing 4. As you may see, all the pointcuts used in the collector aspects don't cover join points within and within control flow of any class defined in package `edu.neu.ccs.mirror.aspects` and its subpackages (package which name has `edu.neu.ccs.mirror.aspects` prefix). Since al the system is defined in the subpackages of `edu.neu.ccs.mirror.aspects`, collector aspects can only reflect system - outside programs but never aspectual reflection implementation.

Listing 4: Any.java

```
package edu.neu.ccs.mirror.aspects.internal;

public class Any {
 pointcut scope():!cflow(within(edu.neu.ccs.mirror
      .aspects..*)) &&
             !within (edu.neu.ccs.mirror.aspects
                   ..*);
 pointcut ConstructorCall():scope() && call(*.new
      (..));
 pointcut ConstructorExecution():scope() &&
      execution(*.new(..));
 pointcut StaticInitialization():scope() &&
      staticinitialization(*);
 pointcut MethodCall():scope() && call(* *.*(..));
 pointcut MethodExecution():scope() && execution
      (* *.*(..));
 pointcut Set():scope() && set(* *.*);
 pointcut Get():scope() && get(* *.*);
 pointcut Initialization(Object created):scope()
       && target(created) && initialization(*.new
      (..));
}
```

4.4.2 *Mirrored Reflection*

Before introducing `Retarget` and `Adapter` modules it is important to note that retargeting client to other source of meta-information implies providing different implementation for Java Core Reflection API. It requires ability to create objects with types, compatible with Java Core Reflection API. However

- Reflection API is implemented as a set of classes and doesn't specify interface that would allow different implementations for the Reflection API;
- All meta-classes in the Reflection API are final and, thereby, cannot be extended.

As a result, only JVM can create meta-objects and different implementation compatible with Java Core Reflection is not feasible.

To solve the problem we provide *mirrored* reflection that mimics Java Core Reflection classes. `java.lang.Class` is mirrored by `edu.neu.ccs.mirror.java.lang.Class` class and classes defined in `java.lang.reflect` package are mirrored with interfaces defined in `edu.neu.ccs.mirror.java.lang.reflect` package. Each of mirrored interfaces corresponds to one of the Reflection classes and provides the same interface to the client as the latter.

As we have mirrored reflection we need to change client's code to use mirrored reflection instead of native one. It is achieved by:

- importing `edu.neu.ccs.mirror.java.lang.Class` class whenever code in the file uses `java.lang.Class` interface, calls `obj.getClass()` or uses `ClassName.class` statement;
- importing `edu.neu.ccs.mirror.java.lang.reflect.*` classe whenever code in the file uses `java.lang.reflect.*` classes interface;
- changing `obj.getClass()` calls to `Class.getClass(obj)` and replacing `ClassName.class` statement with `Class.forName("ClassName")`.

It important to note, that process of redirecting client from Core Reflection API to mirrored reflection needs to be completed before applying `Retarget` aspect to the client's code. It also important to note, that this step is caused only by Core Reflection implementation features and is not conceptually important.

4.4.3 Retaget Aspect

Client can obtain meta-objects only via the following methods of (mirrored) `Class` class:

- `public static Class Class.forName(String,..);`
- `public static Class Class.getClass(Object);`

While first allows to obtain `Class` instance by its name, the second implements run-time type information (RTTI) returning `Class` instance representing type of the `Object` argument. Other meta - objects, such as `Field`, `Method` and `Constructor` could only be obtained `Class` interface.

Following this reasoning, it is enough to redirect executions of the reflection's `Class forName` and `getClass` methods in order to retarget any reflection library client to the new source of meta-information.

`RAspectExecution` aspect, shown at Listing 5 also advises `Class.getEnclosingInstance` method. This method is a mirrored-based extension to the Reflection API and provides clients ability to obtain enclosing object for an instance of a non-static inner class.

Listing 5: RAspectExecution.java

```
package edu.neu.ccs.mirror.aspects.external.
     retarget;
import edu.neu.ccs.mirror.java.lang.Class;
import edu.neu.ccs.mirror.aspects.external.
     reflect.*;

aspect RAspectExecution {

 Object around(String className) throws
      ClassNotFoundException:
  execution(public static Class Class.forName(
       String,..)
   throws ClassNotFoundException) &&
  args(className) {
    return new ClassImpl(className);
 }

 Object around(Object obj):
  execution(public static Class Class.getClass(
       Object)) &&
  args(obj) {
  try{
   return new ClassImpl(obj);
  } catch (Exception e) {
   return null;
  }
 }

 Object around(Object obj):
  execution(
   public static Object Class.getEnclosingInstance
         (Object)) &&
  args(obj) {
   return ClassImpl.getEnclosingInstance(obj);
  }
}
```

As you can see, `RAspectExecution` simply appeals to the `Adapter` module (`ClassImpl` class is a part of `Adapter` module that converts between (mirrored) reflection `Class` and aspectuual repository `Type`).

4.4.4 Adapter

Adapter is a repository-based implementation of a (mirrored) repository API. It is defined within `edu.neu.edu.ccs.mirror.aspects.reflect` package and consists of the following classes:

- `ClassImpl`;
- `MemberImpl`;
- `MethodImpl`;
- `FieldImpl`;
- `ConstructorImpl`.

Each of those classes simply represents wrapper around correspondning repository classes while providing interface specified by mirrored reflection.

4.5 Results

Using only `toString()` and `getModifiers()` methods defined in `java.lang.Class` class we were able to reconstruct most of Reflection API using $\mathcal{AM}$. However, RI$^{\mathcal{A}}$ built is different than Java Core Reflection API in the following features:

- **Read-only structural reflection.** The RI$^{\mathcal{A}}$ implemented doesn't support method invocation and field value setting functionality provided by Java Core Reflection. However, this deficiency is *not* a conceptual one, since the code and data locations in form of signatures are captured by $\mathcal{AM}$ and reflected in the repository. If Java has an ability to to form and send arbitrary message to an object (as part of the language) the problem would be solved;
- **Limited structural static introspection**. The problem essential for a dynamic join point model is a limited reflective capabilities over type structure of a program (relations over

types). $\mathcal{AM}$ allows to reflect program execution while program structural component is revealed only through reasoning over kinds and sequences of join points. For example, `EntityCollector` reveals class hierarchies of a program by analyzing sequences of `initialization` join points in the control flow of a constructor call. Ability to reconstruct `subclass` and `implements` relations over types is based on well-defined protocol specifying order of `initialization` join points occuring on object-creation time. Lack of a similar protocol for class loads disallow to reveal such a knowledge as class is initialized by JVM. Consequences are the follows:

  - Class hierarchies are known only for classes instantiated at least once during the program execution [1]. If there are no instances of a class in the repository then no class hierarchy for the class is known.
  - The only known members for a class are those referenced in the execution path. That is, the method is known to be a member of a class if it was called and/or executed on any instance of this class. Similarly, field is only known to be a member of a class if it was set and/or accessed on any instance of the class.
  - Super/sub type relations over interfaces are not covered since this information is purely static and is not revealed by any dynamic join point. Consequently, meta-information about super/sub interfaces for any given interface is not available.
  - Class hierarchies are only known for a classes actually advised (compiled with the aspect). This limitation is not conceptual but is due to the source-code transformation implementation strategy for AspectJ
  - The aspectual repository reflects essentially non-temporal static meta-information in a temporal manner, i.e. static meta-information adds up as program executes. As a result, the same request to aspectual repository for static data could return different results depending on the point in the execution path the it was made.

- **Full introspection over objects**. Not so surprisingly, an $\mathcal{AM}$ based on a dynamic JPM exposes full object structure of a program. As soon as an object is created by the program, it is reflected in the aspectual repository. Nevertheless, the following limitations exists:

  - Only objects whose classes were actually advised by AspectJ compiler are immediately reflected (on creation-time) in the repository [2].
  - Array objects are never reflected on creation-time since currently AspectJ doesn't capture the corresponding join points. Array objects are only reflected on set/get/method call join points.

- **New features**. Currently, Java Core Reflection doesn't allow to reveal inner/enclosing relations over objects. That is, it doesn't provide enclosing object for an instance of non-static inner class. New method `public static Object Class.getEnclosingInstance(Object)` supported by the aspectual repository supports inner/enclosing object hierarchies.

- **Other new features that can potentially be supported**. The aspectual repository is a database that reflects some JVM internal structures even more precisely than Java Core Reflection does. For example, it has separate meta-objects for classes, interfaces, arrays and primitive types while Core Reflection API unites all of them into `java.lang.Class`. Furthermore, a repository database can answer questions that Core Reflection cannot, for example:

  - Return all objects/classes/interfaces/arrays located thus far in the repository.
  - How many objects of a given class/interface are there?
  - How many times a certain method was executed?

  and so on.

- **Potential support for a services to be provided**. The aspectual repository could also implement a set of listeners of a certain events like object instantiations, method execution, class/interface load etc. Such interfaces would allow broadcasting of AOP events.

- **Security break-in**. Since AspectJ is intentional; uncensored ("unsecured", in the sense of a Java security) $\mathcal{AM}$ and RI$^{\mathcal{A}}$ could easily provide clients with information about private/protected field values, normally inaccessible via Core Reflection.

[1]Class is instantiated if its constructor or constructor of any of its subclasses is called.

[2]This limitation is due to the current implementation of AspectJ and is not conceptual.

5. DISCUSSION

The experiment illustrates that $\mathcal{AM}$ can simulate $\mathcal{RM}$ and provide clients with RI$^{\mathcal{A}}$. Experiment results, however, shows that Hypothesis 1 doesn't strictly hold due to specific features of RI$^{\mathcal{A}}$ as opposed to RI$^{\mathcal{R}}$. At this section we analyze the nature of $\mathcal{AM}$ and compare it with $\mathcal{RM}$. We explore intersection between two and discuss how they can work collaboratively.

5.1 $\mathcal{AM}$ vs $\mathcal{RM}$

$\mathcal{AM}$ has a "behavioral" view of a program. Aspectual mechanism is clearly instruction-oriented since it exposes instruction flow in terms of DA to the clients of its AI. As Figure 7 illustrates, the only structural program meta-information available for DA is the one associated with instructions, available to $\mathcal{AM}$.

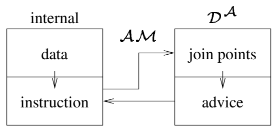

Figure 7: $\mathcal{AM}$

On the other hand, $\mathcal{RM}$ sees a program from the "structural" point of view. As Figure 7 shows, DR reflects program structures.

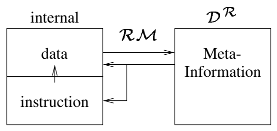

Figure 8: $\mathcal{RM}$

Although having different understanding of a program, mechanisms can simulate each other. The reason why $\mathcal{RM}$ can simulate $\mathcal{AM}$ is that program instructions (i.e. method bodies) are considered by $\mathcal{RM}$ to be the part of the program structure. Consequently, if $\mathcal{RM}$ has write-access for instructions (i.e. it can overwrite methods bodies) it can be used for aspect-weaving, thereby implementing $\mathrm{AI}^{\mathcal{R}}$. However, if $\mathcal{RM}$ is not powerful enough, this opportunity is forbidden. The example is Java Core Reflection, that allows very limited intercession capabilities providing no support for possible $\mathrm{AI}^{\mathcal{R}}$ implementations. Simulating $\mathcal{RM}$ with $\mathcal{AM}$ is more challenging since the only data available to $\mathcal{AM}$ is the data associated with instructions and accessible via join point interface. While few join poins correpond to the program structure in a straightforward manner (such as **set** join point that reveals information about part-of relationship between its target and argument) the others require additional analisys to be mapped on the program structure (for example, `EntityCollector` aspect relies on the order of `initialization` join points to reveal class hierarchy information). This way or the other, $\mathcal{AM}$ can reveal a structural program component through $\mathrm{RI}^{\mathcal{A}}$ only if it is accessible via join point occured in the program execution path. In order words, $\mathcal{AM}$ cannot reflect on structural program components, that are

- Haven't been revealed via join points yet;
- Don't provided by the set of join points supported by AOP language (for instance, **interface** hierarchy is purely structural abstraction that has no effect during program execution);

It explains incompleteness of the aspectial repository for certain categories of meta-information.

5.2 Mechanism Intersection

Ability to simulate one reflective mechanism with the other implies that their domains overlap. Figure 10 illustrates intersection of the internal meta-information regions revealed by the two mechanisms.

In $\mathcal{RM}$, the representation is passive and clients are active: clients must actively pull information. $\mathcal{RM}$ is typically class structure. In $\mathcal{AM}$, the representation is active and clients are passive: clients are notified of or invoked by join points. $\mathcal{AM}$ is typically object behavior (program execution.) $\mathcal{AM}$ includes a JPM, which consists of kinds of join points (event types), and structure of a join point (reflective info about the join point.) The join point structure will typically be callee, caller, args, etc.

5.3 Trade-offs

In the introspection sense, $\mathrm{RI}^{\mathcal{A}}$ is as complete as $\mathrm{RI}^{\mathcal{R}}$, although the lack of statically executable advice in current AOP implementations limits their reflective capabilities. $\mathrm{RI}^{\mathcal{R}}$ and $\mathrm{RI}^{\mathcal{A}}$ have performance-space tradeoffs. MOP and Reflection usually entail considerable performance overhead [6] since they handle (traverse and convert) the internal representation of a meta-information. $\mathrm{RI}^{\mathcal{A}}$ displays improved performance by directly accessing the meta-information via the hash-map. For the same reasons, however, $\mathrm{RI}^{\mathcal{A}}$ is less economical than $\mathrm{RI}^{\mathcal{R}}$, since $\mathrm{RI}^{\mathcal{A}}$ caches everything, it actually duplicates the internal information.

With $\mathrm{RI}^{\mathcal{A}}$, the overhead is adjustable: we have control over what to reflect, i.e., we can reflect only on selected join points (classes, objects, etc). $\mathrm{RI}^{\mathcal{R}}$ is not configurable. $\mathrm{RI}^{\mathcal{R}}$ is available regardless of whether or not it is used. It always uses one source of meta-information (its internal representation) and is tightly coupled to its implementation [16]. In contrast, aspect-based $\mathrm{RI}^{\mathcal{A}}$ can be (un)pluggable allowing better composability. Furthermore, it gives opportunities to increase compile and run-time efficiency by unplugging $\mathrm{RI}^{\mathcal{A}}$ when not needed or not in use.

The join point model provided by AspectJ puts certain limitations on AspectJ-based $\mathrm{RI}^{\mathcal{A}}$. The join point interface does not provide comprehensive static information (e.g., poor reflective abilities over Java interfaces). Furthermore, the `org.aspectj.lang.reflect .MethodSignature` join point interface (which corresponds to the `java.lang.reflect.Method` class in Java) lacks certain meta-method capabilities, e.g., invocation-by-signature (which is provided in $\mathrm{RI}^{\mathcal{R}}$). Nevertheless, the gap between $\mathrm{RI}^{\mathcal{A}}$ and $\mathrm{RI}^{\mathcal{R}}$ is bridgeable. Statically executable advice along with improved lexical a join point model would eliminate these problems.

Degree of overlapping depends on the concrete instances of $\mathcal{RM}$ and $\mathcal{AM}$. While $\mathcal{RM}$ apprehends the program on the level of structural abstractions (i.e. classes, methods, fields, objects etc.) $\mathcal{AM}$ goes further and reflects on the program's instruction stream. MOP, powerful $\mathcal{RM}$ mechanism can be the basis for some $\mathrm{AI}^{\mathcal{R}}$ as it allows to intercept method calls and participate in the dispatch process. However, some instructions are not handled by such a MOP. Consider type-cast expression: (TypeOfExpression) Expression. This expression doesn't relate to any particular structural abstraction. At the same time, native $\mathcal{AM}$ could have a join point, correponding to this kind of expression. In this case, $\mathrm{AI}^{\mathcal{R}}$ would not be able to simulate type-cast join point.

Similarly, $\mathcal{AM}$ could reveal only structural meta-information from the intersection aria. The amount of the structural meta-information covered by $\mathcal{AM}$, and thereby its ability to simulate $\mathcal{RM}$ depends on the maturity of the domains of the AOP language. In the following section we explore relationship between each of domains and expresiveness of the AOP language.

5.4 AOP expressiveness

From $\mathcal{RM}$'s perspective, expressiveness is measured in what data you can read and what data you can write. From $\mathcal{AM}$'s perspective, expressiveness means what instructions you can read, and how you can write instructions. Only certain kinds of instructions are available for $\mathcal{AM}$. Only certain data is connected to the instructions. Only certain intercession are permitted on the instruction stream.

The execution of the program is reflectively represented by $\mathcal{AM}$ in terms of join points. Complete behavioral reflection would be available for $\mathcal{AM}$, if there were a 1:1 mapping between the instruction set of the base language and the kinds of join points, and the join point data reflected the actual arguments to the instructions. The join point stream, however, usually only approximates the execution, since the available kinds of join points do not cover the full instruction set of the base language. For example, AspectJ has a dynamic join point model. The join point model determines the kinds of join points that are related to the instructions of the Java language.

Figure 9 shows the expressiveness space of AOP languages. MCI supports one kind of join point, namely a method-call join point, and the join point interface includes caller, callee, arguments, and result value. MCI has three kinds of advice: dispatch (before call), enter (before execution), and exit (after execution). Aspectj, supports many more kinds of join points. It supports before, after, and around advice. It provides data via `thisJoinPoint` (this, target, results, arguments, etc.), and via `thisJoinPointStaticPart`.

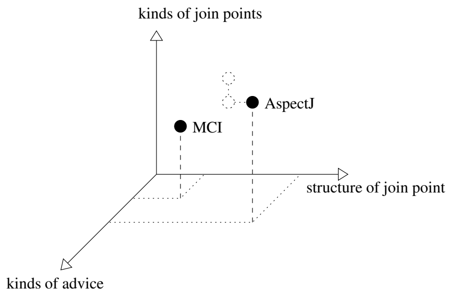

Figure 9: AOP expressiveness model

Often we require static information, for example the declaring class of a method. This kind of information is impossible to acquire from the target object's dynamic information (e.g., actual arguments of the method). Consider AspectJ before `JoinPointStaticPart` was provided. There were two main ways to make this sort of static information available for AspectJ's $\mathcal{AM}$. (1) Add more kinds of join points, e.g., a static method declarations join points. That is, adding a static join point model to AspectJ. This would corresponding to moving the AspectJ dot upwards in Figure 9. (2) Alternatively, extending the join point interface to include also static information, e.g., signature of a method call would also provide required functionality. This would correspond to moving the AspectJ dot more to the right. AspectJ chose the latter by providing the JoinPointStaticPart interface.

5.5 Comparison

Both mechanisms are different. Thus we need both. The reason why one may simulate the other is the partial overlap. Instruction still contain data, and that's how we get a channel to the data. Reflection includes code, and that's how instruction can be rearranged.

The reason why one does not subsume the other is because they don't completely overlap. This fact makes both mechanisms desirable, and sometimes necessary for jobs which neither one can accomplish solely.

Even with full MOP, the lexical information and the internal interpreter instructions are out of reach. For example, reflective information such as source code location (e.g., file name, line number) is available in AspectJ but not in AspectS.

In comparison, $\mathrm{RI}^{\mathcal{R}}$ reflects on classes that were loaded to the JVM [3] therefore accounting for a larger set of classes. Once AspectJ allows byte-code level weaving, advising the `java.lang.ClassLoader` class would eliminate this minor distinction (e.g., using an around advice on the `loadClass` method).

Another approach is to extend the existing AOP model with a mechanism of statically executable advice [13]. Such a mechanism would furnish a complete picture of the class structure statically. $\mathrm{RI}^{\mathcal{A}}$ could then expose more structural meta information than $\mathrm{RI}^{\mathcal{R}}$ does today.

[3]With Java Core Reflection, one cannot reflect on classes until they are loaded to the JVM.

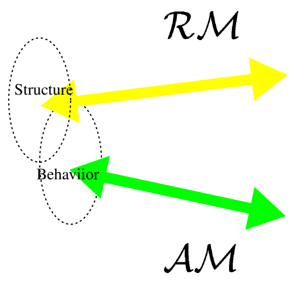

Figure 10: intersect

When we have a non-native $\mathcal{AM}$, it is restricted to capabilities of the base language reflective capabilities (when implemented on top of MOP or using program code transformation). At the same time native implementation can use internals of interpreter and have more control over exposed information and could have richer set of join points.

Obviously, $\mathcal{AM}$ stems from $\mathcal{RM}$. It mimics $\mathcal{RM}$ hooks-meta-object collaborative mechanism. On the other hand, $\mathcal{AM}$ goes beyond $\mathcal{RM}$ by generalizing meta-object term to aspect. Both are definitely located at meta-level and feed on behavioral program meta-data. The difference is "cross-cutting" nature of aspects. In other words, while the meta-object has a corresponding item from within its base level entity structure (class or object), aspect doesn't have one. Instead, it (the aspect) is connected to program execution (behavioral meta-information) given in terms of (sets of) join points. Similarly, the join point is a generalization of MOP's "hooks". While hook has well-defined purpose in MOP protocol (it connects certain points in program base level structure to certain meta-objects), join point is defined only in terms of base program - it could have or could not have any aspect, or could have set of aspects associated with it.

$\mathcal{RM}$ views the program statically, as a database of structural components. Since this database also includes the program's code (set of instruction) $\mathcal{RM}$'s intercession capabilities over code component could easily change program's behavior.

$\mathcal{AM}$ considers the program in terms of a program run-time. Basically, $\mathcal{AM}$ is unaware about the program structure, but rather well-aware of program execution path, detailed to a join point stream. Therefore, $\mathcal{AM}$ could only introspect on structure by assumptions made over join point stream (if given rich enough join point model - join point set and (reflective) structure of a join point) and is not able to intersect on structure. But it's view of program execution is amazing. It has "full" (as JPM allows) view of program behavior and rich (again, as ADM allows) intercession capabilities on it.

$\mathcal{AM}$ and $\mathcal{RM}$ should complement each other if complete reflective capabilities is needed. (Probably, they should evolve to become one core mechanism). AOL with statically executable advice is an example of one system that has dynamic JPM reflective capabilities over program behavior and "full" static view of a program. But still, it would lack direct mechanism to introspect on structure MOP, on the other hand, has full structural control and allows to implement AOP on top of it if behavioral introspection is necessary. But, again, MOP should be COMPLETE enough to implement all the interesting join points.

From a software engineering perspective, AspectJ has limited aspectual (incomplete static information) and reflective capabilities (bound by Java Core Reflection), which limits its area of applicability. In the rest of this Section, we explore opportunities to overcome these limitations by combining aspectual and reflective features of AspectJ in unexpected ways.

5.6 Reflection within advice

Compile-time weaving promoted by AspectJ in contrast with AspectS dynamic weaving approach gives better run-time performance though less flexibility. Using RI$^{\mathcal{R}}$ within AspectJ's advice could improve aspects run-time flexibility.

An example below illustrates this idea. The `Notifier` aspect, shown at the listing invokes a static no-argument methods (listeners) using RI$^{\mathcal{R}}$ method-invocation technique. The basis for a method selection is equality between method name and the name of the join point target class. Notice, that aspect `Notifier` knows nothing about `ClassListeners` class (or even if such a class exists).

Further development of this approach would allow dynamically (de)attach functionality to advised join points providing limited run-time aspects weaving [5, 2].

However, it is possible to get the same functionality as the example provides without using RI$^{\mathcal{R}}$. But that would entail a dedicated aspect and advice for each method resulting in a loss of a dynamic flexibility.

```
class ClassObserver {
  static void A() {
  System.out.println("A Listener");
  }
  static void C() {
  System.out.println("C Listener");
  }
}
```

```
aspect Notifier {
  before(Object targ): within(!Notifier)
  && call(* *(..)) && target(targ) {
  try {
    String className =
    targ.getClass().getName();
    Class.forName("ClassListeners")
    .getDeclaredMethod(className,null)
      .invoke(null,null);
  } catch (Exception e) {}
 }
}
```

5.7 Generative aspects

Some tasks that rely on structural meta-information cannot be done with either RI$^{\mathcal{R}}$ or AI$^{\mathcal{A}}$[4]. Consider a lexical style guideline for OO program coding, which states that call-sites within a method should only target instance-variable classes of the enclosing class and argument classes of the method. This style rule is a simplified variant of the Class Form of the Law of Demeter [14], and detecting violations of this rule is polynomial and requires only static structural metal-information. The reason the style rule cannot be checked with only RI$^{\mathcal{R}}$ is that it does not expose call-sites within method code. In a related AOSD'03 paper [13] we also prove that it is impossible to check such rules in AspectJ. Specifically, we consider the following aspect:

[4] In the context of Java/AspectJ

Listing 7: A.java

```
class A {
 C c=new C();
 B b=new B();
 void foo() {
  foo();
  c.foo();
  b.foo(c);
  D d=new D();
  d.foo(); // style rule violation
 }
}
```

```
abstract aspect Violation {
  abstract pointcut Violation;
  declare warning: Violation: "Violation";
}
```

where `Violation` refers to the style rule violation. Indeed, there is no way to avoid an **`if`** pointcut primitive in the pointcut designator instantiating `Violation` for the above style rule. But the **`if`** pointcut primitive is not statically determinable and therefore inappropriate in for the **`declare warning`** construct.

It is also impossible to perform a complete check dynamically, because at run-time not all class meta-information is available; and, moreover, not all method calls are covered. This holds independently of whether we use RI$^{\mathcal{R}}$ within the aspect or not.

However, reflection can be employed to generate an aspect that *does* implement the `Violation` aspect successfully using the `generateAspect` method shown in Listing 6. Listing 8 shows the generated aspect for a particular class hierarchy: the code listings for class `A` is shown in Listing 7; classes `B`, `C`, `D`, and interface `I` are omitted.

Listing 6: Aspect generator

```
public void generateAspect(StyleRule rule) {
  FileWriter writer = null;
  try{
  writer = new FileWriter(fileName);
  writer.write("aspect LoDViolation extends
       Violation {\n");
  List declarations = rule.getPointcutDeclarations
       ();
  for (int i=0;i<declarations.size();i++)
    writer.write((String)declarations.get(i)+"\n")
       ;
  writer.write("pointcut LoD(): "+rule.
       getRulePointcut()+";\n");
  writer.write("pointcut Violation(): !LoD();\n");
  writer.write("}");
  } catch(Exception e) {
  e.printStackTrace();
  } finally {
  try {writer.close();} catch(Exception exc) {}
  }
}
```

Listing 8: LoDViolation.java

```
aspect LoDViolation extends Violation {
  pointcut Global(): within(*) && call(* *.*(..));
  pointcut A(): within(A) && call(* (!(B || C))
       .*(..));
  pointcut A_foo(): withincode(* A.foo()) && call
       (* (!(A)).*(..));
  pointcut B(): within(B) && call(* (!(I)).*(..));
  pointcut B_foo_C(): withincode(* B.foo(C)) &&
       call(* (!(B || C)).*(..));
  pointcut C(): within(C) && call(* *.*(..));
  pointcut C_foo(): withincode(* C.foo()) && call
       (* (!(C)).*(..));
  pointcut D(): within(D) && call(* *.*(..));
  pointcut D_foo(): withincode(* D.foo()) && call
       (* (!(D)).*(..));
  pointcut LoD(): (Global() && ((A() && (A_foo()))
        || (B() && (B_foo_C())) || (C() && (C_foo()
       )) || (D() && (D_foo())))));
  pointcut Violation(): !LoD();
}
```

6. RELATED WORK

In this section we give a brief overview of research conducted in areas of reflection and AOP and state how these works related to our. We describe the related work using the terminology presented in this paper in order to make the comparison clearer.

Related work are classified into two main groups:

- Works conducting research and experiments in computational reflection. Specifically, three approaches are considered:
  - A reflection-based approach to AOP (i.e., AOP as a facilitation for MOP).
  - A transformation-based approach to AOP.
  - Pluggable reflection [16].
- Works in AOP semantics that take a step toward recognizing AOP-expressiveness as being first-class in the syntactical and semantical domains of programming languages. By providing semantics for AOP functionality, these works describe a native $\mathcal{AM}$, thereby decoupling AOP from reflection-based and transformation-based approaches.

6.1 Reflection-based approach to AOP

The connection between AOP and reflection is usually examined through the conceptual framework of MOPs. Works overviewed in this section are no exception. The goal in these works is different than ours. However, a close examination of the implementation techniques used to build reflection-based $\mathcal{AM}$s also highlights the overlap between $\mathcal{AM}$ and $\mathcal{RM}$, which is the focus of our study.

6.1.1 Aspect-Oriented programming using Reflection.

Sullivan [18] describes the development of a reflective object-oriented language (with MOP) for supporting AOP extensions. More specifically, AI (a syntactic extension) is introduced to Java by extending the base language type system with predicate types that include predicate and advice expressions. The extended type system then allows to affect the method-dispatch process by evaluating advice expressions of predicate-type variables which predicate-match the method-invocation properties (predicate could have dynamic or static parts).

Sullivan's work [18] highlights the intersection between $\mathcal{AM}$ and $\mathcal{RM}$. It studies what features should $\mathcal{RM}$ support in order to provide proper semantics for AI. For example, the semantic of advice is given in terms of a (meta-level) virtual function dispatch method. Sullivan concludes that $\mathcal{RM}$ with a rich introspective and intercessive capabilities over the program structure and behavior gives sufficient basis for an $\mathcal{AM}$ implementation to support the corresponding AI$^{\mathcal{R}}$.

6.1.2 Two-Step weaving with Reflection using AspectJ

An interesting example of $\mathcal{RM}$—$\mathcal{AM}$ collaboration in Java/AspectJ is given by Ledoux et al. [5]. As opposed to Sullivan's observations, Ledoux et al. shows how certain MOP functionality not found in Java could be simulated by $\mathcal{AM}$ in AspectJ. A proxy-based runtime MOP system, called RAM (Reflection for Adaptable Mobility), is built on top of both Java's $\mathcal{RM}$ and AspectJ's $\mathcal{AM}$.

RAM could be seen as RI$^{\mathcal{A}}$ (at least, partially), but there are dramatic difference in the point of view between Ledoux et al.'s work and ours. While employing $\mathcal{AM}$ to provide an essential piece of RAM functionality ([5, section on AOP/Reflection relationship]), the authors view AOP as facilitation for MOP and conclude on similarities between MOP hooks and AOP join points and on the similarities between meta-objects and aspects. We, on the other hand, emphasize the other direction.

Nevertheless, overall the system demonstrates that a collaboration between $\mathcal{AM}$ and a read-only structural $\mathcal{RM}$ (Java Core Reflection) could raise into a fully-functional $\mathcal{RM}$ (RAM). Consequently, we consider this as another example illustrating that $\mathcal{AM}$ provides a strong alternative to $\mathcal{RM}$ introspection and to the intercession behavioral component.

6.2 Transformation-based approach to AOP

The reflective information required to implement AOP could be obtained from the program's source code. $\mathcal{AM}$ can then be implemented by weaving AOP-expressiveness into the base-level program at compile-time (or load time for byte-code weaving) via source code instrumentation thereby guaranteeing the causal connection at run-time. AspectJ is an example of this approach. The transformation-based in AspectJ illustrates the behavioral nature of $\mathcal{AM}$ as reflective mechanism, since it is AOP functionality is achieved by directly changing the original instruction stream (source code).

6.3 Pluggable reflection

Reflective abilities of a programming language are typically tightly coupled with the language implementation (built into the interpreter to guarantee causal connection), and tightly coupled with the reflective programs written in this language (i.e., reflective computation is a client and provider of the meta-information). Elsewhere [16] we preset a different approach that views reflection as a programming interface that could be separated from it's implementation and from clients. As an example, an implementation of a mirrored reflection interface that uses a source code repository is given. There is a strong connection between the works (e.g., here we also provide a mirrored reflection to clients).

Both works put forward the lesson that a language's native reflection mechanism, namely $\mathcal{RM}$, is not the only possible provider of computational reflection. While in [16] we substitute Java Core Reflection with a source code repository, in this paper we simulate it using $\mathcal{AM}$. Note that a repository-based reflection has a structural introspective nature. An interesting followup research would be to see if the repository-based reflection could be joined with AOP and what reflective capabilities this mutated AOP would have (e.g., statically executable advice [13]).

6.4 AOP semantics

Models for allowing native $\mathcal{AM}$ implementation are also addressed from the programming language perspective. In this section we give an overview two works on AOP-semantics.

6.4.1 *Method-Call Interception*

Method-Call Interception [12] (MCI) is an extension to a simple core Object-Oriented language μO^2 defined (which is also defined that paper.) MCI is presented for addressing issues related to attaching additional functionality to method calls. The work provides syntax, static and dynamic semantics for MCI expressiveness in terms of the language domains. In order words, MCI defines both an AI (syntax) and a native $\mathcal{AM}$ for the μO^2 language. The fact that the extended μO^2 has an $\mathcal{AM}$ but doesn't have a native $\mathcal{RM}$ serves as proof that the two are independent mechanisms.

In the context of our research, the MCI example raises the following questions:

- Can we provide $\text{RI}^{\mathcal{A}}$ in the MCI-extended $lang$? If so, to what extend? If not, why?
- What JPM and what ADM should the AOL have to allow implementation of $\text{RI}^{\mathcal{A}}$?
- What features would $\text{RI}^{\mathcal{A}}$ have?

These questions could motivate research aimed to implementing $\text{RI}^{\mathcal{A}}$ in MCI (or in an extended MCI) language. We expect this kind of work to improve our understanding of aspectual reflection and AOP as a whole.

6.4.2 *Semantics for Dynamic Join Points*

Denotational semantics for a functional AOP mini-language is given in Wand et al. [19]. Although the model introduced is based on functional language, it is quite similar to AspectJ.

The work examines the semantics for JPM and ADM. The JPM and ADM studied are more mature compared with those in MCI. In pursuing the followup research proposed above, Want et al.'s work [19] would play a role a provider of an alternative native $\mathcal{AM}$. One idea is to extend MCI's JPM and ADM with the best features from both AOP-semantics worlds.

7. CONCLUSION

In a nutshell, this paper heightens the awareness that $\mathcal{RM}$ and $\mathcal{AM}$ are alternative providers of computational reflection (Figure 11). Both $\mathcal{RM}$ and $\mathcal{AM}$ exhibit aspectual and reflective features. AOSD approaches differ in the features they use. AspectS combines the aspectual and reflective features of the Smalltalk MOP. AspectJ combines the reflective features of Java with new aspectual features.

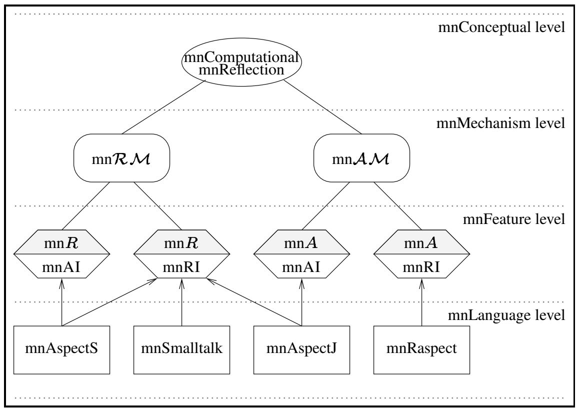

Figure 11: Computational Reflection Space

The leftmost feature-level hexagon in Figure 11 denotes the implementation of an aspectual interface over reflection, namely $\text{AI}^{\mathcal{R}}$. The rightmost hexagon in Figure 11 denotes the implementation of a reflection API over AOP, namely $\text{RI}^{\mathcal{A}}$. The two center hexagons, $\text{RI}^{\mathcal{R}}$ and $\text{AI}^{\mathcal{A}}$, represent the traditional implementation of reflection and AOP.

Figure 11 also maps the space of AOSD languages. Pure OOP languages, like Smalltalk, utilize just the reflective features of $\mathcal{RM}$. Analogously, a pure AOP languages would be one that utilizes just aspectual features. In the figure, we propose a novel *Raspect* language, which utilizes only the reflective features of $\mathcal{AM}$.

We have identified, analyzed, and provided examples of explicit aspectual support in $\mathcal{RM}$ and explicit reflective support in $\mathcal{AM}$, and illustrated AOP and Reflection collaboration from a practical software engineering perspective. Understanding the software engineering trade-offs could also impact future language design and implementation.

In a keynote talk on *Reflection, MOPs, AOP and back again for more?* [5], Kiczales explained the distinction between AOP and reflection, and suggested that seeds of other (good) ideas like AOP might be found in reflection. When a native $\mathcal{AM}$ becomes a reality, and reflection is implemented as just another aspect, we can search for seeds of other (good) ideas like reflection in AOP.

8. REFERENCES

[1] AOSD 2002. *Proceedings of the 1st International Conference on Aspect-Oriented Software Development*, Enschede, The Netherlands, Apr. 2002. ACM Press.

[2] J. Baker and W. Hsieh. Runtime aspect weaving through metaprogramming. In AOSD 2002 [1], pages 86–95.

[3] J. Brant, B. Foote, R. E. Johnson, and D. Roberts. Wrappers to the rescue. In E. Jul, editor, *Proceedings of the 12th European Conference on Object-Oriented Programming*, number 1445 in Lecture Notes in Computer Science, pages 396–417, Brussels, Belgium, July 20-24 1998. ECOOP'98, Springer Verlag.

[4] S. Chiba. Load-time structural reflection in Java. In E. Bertino, editor, *Proceedings of the 14th European*

[5] Reflection 2001.

Conference on Object-Oriented Programming, number 1850 in Lecture Notes in Computer Science, pages 313–336, Cannes, France, June 12-16 2000. ECOOP 2000, Springer Verlag.

[5] P.-C. David, T. Ledoux, and N. M. N. Bouraqadi-Saâdani. Two-step weaving with reflection using aspectj. In *Proceedings of the 16th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, Tampa Bay, Florida, Oct. 14-18 2001. OOPSLA'01, ACM SIGPLAN Notices 36(11) Nov. 2001.

[6] Y.-G. Guéhéneuc. Overall impression on the AOP workshop. `http://www.yann-gael.gueheneuc.net/Work/Publications/Documents/Trip+rep%ort+AOP+Workshop01.doc.pdf`, May 2001.

[7] R. Hirschfeld. AspectS—aspect-oriented programming with Squeak. In M. Aksit, M. Mezini, and R. Unland, editors, *Architectures, Services, and Applications for a Networked World*, number 2591 in Lecture Notes in Computer Science. Springer Verlag, 2003.

[8] G. Kiczales. Beyond the Black Box: Open Implementation. *IEEE Software*, 13:1:8, 10–11, January 1996.

[9] G. Kiczales, J. des Rivires, and D. G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.

[10] 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.

[11] S. Kojarski, K. Lieberherr, D. H. Lorenz, and R. Hirschfeld. Aspectual reflection. In *AOSD 2003 Workshop on Software-engineering Properties of Languages for Aspect Technologies*, Boston, Massachusetts, Mar.18 2003. AOSD 2003 Workshop on Software-engineering Properties of Languages for Aspect Technologies, ACM Press.

[12] R. Lämmel. A semantical approach to method-call interception. In AOSD 2002 [1], pages 41–55.

[13] K. Lieberherr, D. H. Lorenz, and P. Wu. A case for statically executable advice: Checking the Law of Demeter with AspectJ. In *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development*, pages 40–49, Boston, Massachusetts, Mar. 17-21 2003. AOSD 2003, ACM Press.

[14] K. J. Lieberherr, I. Holland, and A. J. Riel. Object-oriented programming: An objective sense of style. In *Proceedings of the 3rd Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 323–334, San Diego, California, Sept. 25-30 1988. OOPSLA'88, ACM SIGPLAN Notices 23(11) Nov. 1988.

[15] C. V. Lopes and G. Kiczales. Recent developments in AspectJ. In S. Demeyer and J. Bosch, editors, *Object-Oriented Technology. ECOOP'98 Workshop Reader*, number 1543 in Lecture Notes in Computer Science, pages 398–401. Workshop Proceedings, Brussels, Belgium, Springer Verlag, July 20-24 1998.

[16] D. H. Lorenz and J. Vlissides. Pluggable reflection: Decoupling meta-interface and implementation. In *Proceedings of the 25th International Conference on Software Engineering*, pages 3–13, Portland, Oregon, May 1-10 2003. ICSE 2003, IEEE Computer Society.

[17] P. Maes. Concepts and experiments in computational reflection. In *Proceedings of the 2nd Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 147–155, Orlando, Florida, Oct. 4-8 1987. OOPSLA'87, ACM SIGPLAN Notices 22(12) Dec. 1987.

[18] G. Sullivan. Aspect-oriented programming using reflection and metaobject protocols. *Communications of the ACM*, 44(10):95–97, 2001.

[19] M. Wand, G. Kiczales, and C. Dutchyn. A semantics for advice and dynamic join points in aspect-oriented programming, 2002.