Proceedings of the Second workshop on Virtual Machines and Intermediate Languages for emerging modularization mechanisms (VMIL 2008), held at the Twenty-third ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, October 21-23, Nashville, Tennessee, United States

VMIL '08

Workshop Chair:
Hridesh Rajan

The Association for Computing Machinery
1515 Broadway
New York New York 10036

ACM ISBN: 978-1-60558-384-6

Workshop Organization

Program Committee:

- Christoph Bockisch, Darmstadt University of Technology
- Eric Bodden (McGill University, Canada)
- Juan Chen (Microsoft Research, USA)
- Shigeru Chiba (Tokyo Institute of Technology, Japan)
- Sophia Drossopoulou (Imperial College, UK)
- Robert Dyer (Iowa State University, USA)
- Eric Eide (University of Utah, USA)
- Matthew Flatt (University of Utah, USA)
- Michael Haupt (Hasso Plattner Institute, University of Potsdam, Germany)
- Gregor Kiczales (University of British Columbia, Canada)
- Hidehiko Masuhara (University of Tokyo, Japan)
- Greg Morrisett (Harvard University, USA)
- Angela Nicoara (ETH Zurich, Switzerland)
- Harold Ossher (IBM Research, USA)
- Hridesh Rajan, Iowa State University
- Don Syme (Microsoft Research, UK)

Organization Committee:

- Hridesh Rajan, Iowa State University
- Christoph Bockisch, Darmstadt University of Technology
- Michael Haupt (Hasso Plattner Institute, University of Potsdam, Germany)
- Robert Dyer (Iowa State University, USA)

Invited Talks

Memory Management for Hard Real-time Systems

Jan Vitek
Purdue University

Abstract:
The Java programming language has become a viable platform for real-time systems with applications in avionics, shipboard computing, audio processing, industrial control and the financial sector. High performance real-time Java virtual machines (RT JVMs) are now available from multiple vendors.

One of the main challenges in using a high level programming language, such as Java or C#, to program hard-real time system is to deal with heap-allocated data structures. Traditional techniques such as pre-allocation and object pooling are ill-suited to modern software engineering practices. In this talk I describe two approaches that we have experimented with in the context of the Ovm real-time Java virtual machine: region-based allocation and real-time garbage collection. I will demonstrate that for tasks which can tolerate latencies on the order of 1 millisecond real-time collectors are perfectly adequate, but, in order to obtain sub-milliseconds latencies other approaches are required. The talk will also give an overview of new results in non-blocking concurrent garbage collection.

About the Speaker:
Jan Vitek is an Associate Professor in Computer Science at Purdue University. He leads the Secure Software Systems lab. He obtained his PhD from the University of Geneva in 1999, and a MSc from the University of Victoria in 1995. His research interests include programming language, virtual machines, mobile code, software engineering and information security.

Compiling the Web--Building a Just-in-Time Compiler for JavaScript

Andreas Gal
Mozilla Corporation

Abstract:
Over the last decade we have made great strides towards improving the execution performance of virtual-machine (VM) based high level programming languages. Today, dynamic compilation is standard in most Java and C# VMs, enabling programs written in these languages to execute with similar efficiency as legacy type-unsafe C or assembly code.

However, for the past decade much of the research and development effort in the Just-in-Time compilation domain was focused on the runtime compilation of statically typed languages, leaving an important and steadily growing field of programming languages behind: dynamically typed high level languages such as JavaScript, Python, PHP and Ruby. Combined with an explosive growth of web applications and the wide-spread use of dynamically typed programming languages for the client and server side of such web applications, this has created a situation where bytecode interpretation is suddenly again the predominant mode of execution for much of the code used on a daily basis on desktop computers, including popular web programs and services like Google Mail or Google Docs.

In this talk I will report on the design and development of TraceMonkey, the JavaScript Just-in-Time compiler in Mozilla's Firefox web browser. I will discuss the fundamental differences between statically typed and dynamically typed languages from the perspective of a compiler constructor, and I will highlight some of the unique challenges for compiling dynamically typed languages such as JavaScript.

About the Speaker:
Andreas Gal is a Project Scientist at the Computer Science Department of the University of California, Irvine. He is currently on leave, working with Mozilla on TraceMonkey, a Just-in-Time compiler for JavaScript. Andreas received his PhD from the University of California, Irvine, in 2006. His research interests include virtual machines, dynamic compilation, programming languages and mobile code.

Liquid Metal: Blurring the Hardware/Software Boundary

David F. Bacon
IBM Research

Abstract:
The goal of the Liquid Metal project is to allow a heterogeneous system of conventional processors and reconfigurable hardware (FPGAs) to be programmed in a single language with transparent, dynamic execution across the aggregate computing resources -- to "JIT the hardware". Achieving this goal requires significant innovation across the entire system: language design, compiler technology, hardware synthesis, the run-time system, and hardware protocols. I will give an overview of the Liquid Metal language and the tool chain we have built, present some initial results, and describe challenges for the future.

About the Speaker:
David F. Bacon is a Research Staff Member at IBM's T.J. Watson Research Center. He leads the Metronome project which pioneered hard real-time garbage collection, opening the use of high-level languages like Java for time-critical systems in financial trading, aerospace, defense, video gaming, and telecommunications.

Dr. Bacon's algorithms are included in most compilers and run-time systems for modern object-oriented languages, and his work on Thin Locks was selected as one of the most influential contributions in the 20 years of the Programming Language Design and Implementation (PLDI) conference. His recent work focuses on high-level real-time programming, embedded systems, programming language design, and reconfigurable hardware. He received his Ph.D. in computer science from the University of California, Berkeley and his A.B. from Columbia University. He holds 7 patents and has served on numerous program committees including POPL, OOPSLA, ECOOP, LCTES, and EMSOFT. He is a member of the IBM Academy of Technology, Distinguished Scientist of the ACM, and is on the governing boards of ACM SIGPLAN and SIGBED.

Table of Contents

Composing New Abstractions From Object Fragments

Adrian Kuhn and Oscar Nierstrasz

Software Composition Group
University of Bern,Switzerland
{akuhn, oscar}@iam.unibe.ch

Abstract

As object-oriented languages are extended with novel modularization mechanisms, better underlying models are required to implement these high-level features. This paper describes CELL, a language model that builds on delegation-based chains of object fragments. Composition of groups of cells is used: 1) to represent objects, 2) to realize various forms of method lookup, and 3) to keep track of method references. A running prototype of CELL is provided and used to realize the basic kernel of a Smalltalk system. The paper shows, using several examples, how higher-level features such as traits can be supported by the lower-level model.

1. Introduction

In order to extend object-oriented languages with new means of abstraction, better intermediate representations are required (16). Current approaches typically flatten new abstractions down to existing concepts (9; 6; 15). This is mainly due to limitations of current intermediate models that cannot represent other modularization mechanisms than objects and classes. Flattening of abstractions limits their usefulness. For example, it is often hard, if not impossible, to manipulate flattened abstractions at runtime. Sometime, even meta-information about the abstractions is stripped away and thus not available at runtime, so that many runtime optimizations cannot take these abstractions into account (5). And worst of all, the flattening process might cause existing abstractions to impose severe restrictions upon the new abstractions and thus render them partially useless. For example, Java's erasure of generics is well known for introducing several such flaws.

Sometimes language extensions are realized using language hooks, such as the *method_missing* hook (2; 10; 13) which is often found in dynamic languages. Given that these hooks are typically exposed as methods that must be overridden, their usefulness in a practical setting is limited. For example, consider two libraries that both override the same hook method. The second library that loads will override and thus uninstall the hook installed by the first library, putting the entire system in an unspecified state. Obviously, language extension points that provide better means of modularization than a simple hook method are required.

A third, seemingly unrelated motivation for this work is better means for analysis and reverse engineering of running applications. However, as these tasks often require the introduction of novel abstractions, they are clearly related to the above motivations. Consider for example a back-in-time debugger (11). To implement such a tool, novel abstractions such as persistent objects (17) and aliases (12) must be introduced.

This paper describes CELL, an intermediate language model that breaks objects into a "sea of fragments" (16). Each object, each class, and any other modularization mechanisms, is represented as a group of one or more collaborating cells. The CELL model is message-based (20) and uses delegation-based cell composition to realize method lookup as in the Aspect Machine (8). Everything is a cell. There is no asymmetric distinction that separates actual objects and mere fragments. Rather, being an object is an emergent property of collaborating cells—sometimes with sharp boundaries, sometimes blurring into each other and thus giving rise to novel modularization mechanisms.

As a proof of concept, we provide a running prototype of CELL. The prototype is written in Java and Python and has been deliberately limited on first getting the concepts right, before any performance optimizations are to be considered. When running a cell-based system with the prototype, the following layers are present:

- C3, a running program in the high-level language,
- C2, the kernel of the high-level language, done using the CELL model. We refer to this as a cell-based kernel,
- C1, the kernel of CELL, currently implemented in Java.

In this paper, cell-based kernels of class-based languages are investigated. We present CELLTALK, a basic Smalltalk sys-

VMIL'08, October 19, 2008, Nashville, TN.

tem[1]. Celltalk is further extended with additional language features: frist traits (19), then object-references (12). We show how this is done purely in terms of the Cell model.

The remainder of this paper is structured as follows: Section 2 introduces the CELL model. Sections 3 to 5 present examples of increasing complexity, cumulating in the presentation of CELLTALK in Section 6, which is extended with traits in Section 7, and with aliases in Section 8. Current limitations of CELL are discussed in Section 9, related work in Section 10, and Section 11 concludes.

2. The Language Model of CELL

The language model that we describe here introduces an additional layer of abstraction *below* objects, to be used as an intermediate model for representing and implementing object-oriented languages. The key motivation of the described language model is to support emerging modularization mechanisms such as aspects, traits, or *method_missing* hooks.

The basic building blocks of the CELL model are object fragments (16) which we refer to as *cells*. Objects are built up from groups of cells with delegation relationships between them (8). The CELL model is message-oriented (20), so all interaction between cells is based on message sending. Messages and message replies are also cells, thus "everything is a cell" .

Cells themselves are simple structures, mainly consisting of

- a lookup function,
- a delegation pointer[2], and
- an (optional) payload.

Cell composition can be used to realize various forms of method dispatch, including novel modularization mechanisms. Compared to delegation-based AOP in the Aspect Machine (8), delegation in CELL is associated to a pointer instead of a delegate function. On the other hand, the lookup function of CELL is more powerful, taking into account all information provided by a `Message` cell and not just the message name.

The payload of a cell can either be binary data (used to store information such as integer values, *etc.*) or a list of references to other cells. The payload of each cell instance is private to that instance and can only be exposed through the means of primitive functions that act as closures over the payload.

[1] The choice of Smalltalk as a running example for this paper has been rather arbitrary and was mainly motivated by the author's proficiency with Smalltalk's object model. We could have chosen, without loss of generality, almost any other class-based language, such as Ruby, Python, *etc.*

[2] In the remainder of the paper, we use the terms "delegation pointer" and "`next`" interchangeably, and thus often refer to the delegate of a cell as "the next cell (after that cell)".

The current prototypes of CELL are written in high-level languages (Java and Python), as we decided to first get the concepts right rather than to strive for optimal performance up-front. So far, we identified the following types of cells that seem likely to be used in most language realizations:

- `Alias` and `Head` cells serve to handle identity issues. Head cells delegate only; they have no payload and do not respond to any messages. Head cells are most useful as anchor points for objects (or other modularization mechanisms) by providing a cell identity that can be used as the object's identity as well (16). As such, head cells serve the same purpose as object proxies do in the Aspect Machine (8). Alias cells, on the other hand, are a refinement of head cells used to keep track of object references (12). In Section 8 we show an example that makes use of alias cells.
- `Slot` cells respond to two messages, a setter and a getter message, and contain a reference to another cell, the value of the slot. Slot cells have a predecessor in Self's slots (21), which also have two implicit messages.

  The actual naming convention of the accessor messages is up to the implemented language. In this paper, we refer to slot names using an at sign (`@`) in order to distinguish them from function names, which start with a hash (`#`).
- `Function` cells, where the payload consists of a string and a callable cell. If the name of a sent message matches the string, the callable cell is executed.
- `WInteger` cells, where the payload consists of the binary data of an integer number. Responds to a common set of primitive operations on the integer payload.
- `WString` cells, where the payload consists of the binary data of a string. Responds to a common set of primitive operations on the integer payload.
- `WArray` cells, where the payload consists of a variable-sized array of references to other cells. Responds to a common set of primitive operations on the array payload.
- `CustomLookup` cells raise lookup from the level of the intermediate language to the high-level language. Upon lookup, another cell (that must respond to `#lookup`) is used to carry out the actual lookup.

  Custom-lookup is supposed to replace the *method_missing* hook as it is found in many dynamic languages. The advantage of custom-lookup is that it offers a modular extension mechanism rather than a single point of extension. For example, using custom-lookup multiple libraries can extend the same class without interfering with each other.
- `Branching` cells have *two* delegation pointers. Lookup is delegated to both delegates: first to the branch, and afterward to the next cell. Branches can easily be mapped to

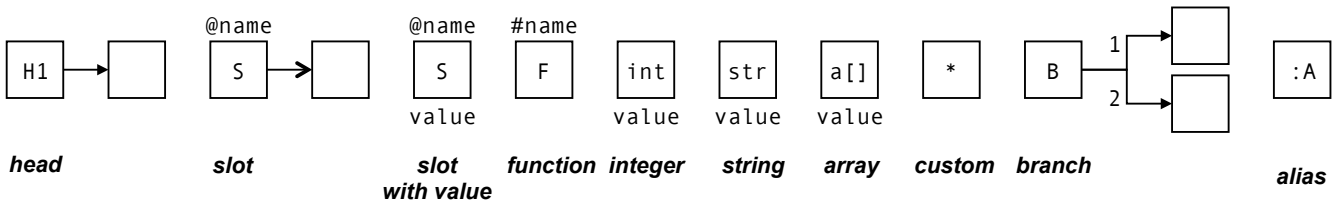


Figure 1. Notation of cell instances: (from left to right) a `Head` cell with delegation arrow; a `Slot` cell with a reference arrow, above the name of the slot; another `Slot` cell, above again the name, below the primitive value stored in the slot (reference arrow and primitive value obviously exclude each other); a `Function` cell, above the name of the function; an `WInteger`, a `WString`, an `WArray` cell, below the value of each; an `CustomLookup` cell; a `Branch` cell with branches b_1 and b_2; and an `Alias` cell (see Section 8).

multiple delegates, as in the Self programming language (21).

- `Message` cells, where the payload consists of a receiver, a name, an array of arguments, and an integer which indicates the "order of resend" (see Subsection 2.1). Message cells are used when sending messages from one cell to another (20).
- `Callable` cells that can be executed by the underlying CELL machine. Some callables have an optional payload, that is, they are closures. Callable cells are typically returned upon successful lookup of a message. Some callables close over the private payload of the returning cell.

Furthermore, the same three cell instances are globally used whenever we need to refer to `nil`, `true`, and `false`. They will typically delegate to further cells that will provide functionality required for these datatypes in the realized high-level language.

When drawing a diagram of cell instances, we use the symbols presented in Figure 1. Cell instances are represented by squares that are marked with letters to denote different cell types. Different kind of arrowheads are used to indicate delegation and reference. Please refer to the caption of Figure 1 for details.

2.1 On Lookup of Messages

The CELL model is message-oriented (20), as such, all interaction between cells is based on message sending. The lookup mechanism iterates over the delegation tree of a receiver and stops when a cell in the tree responds to the sent message.

Resending messages is treated specially: a resend starts at the receiver again and uses "response counting" to skip responses. For example, given a message `#foo` with a resend order of r=2, lookup does not return the first response, but rather the second response from the tree of visited cells.

The lookup mechanism is implemented using the following set of kernel primitives (listed here as UML method signatures), for which a Python implementation is given in Appendix A:

```
Callable.call(Message): Cell
Cell.accept(Visitor)
Cell.full_lookup(Message): Callable
Cell.lookup(Message): Callable
Cell.send(Message): Cell
Lookup.visit(Cell)
Message.resend(): Cell
```

The lookup mechanism works as follows:

1. A cell S sends a message (consisting of name and arguments) to cell R. We refer to S as the *sender* and to R as the *receiver*.
2. A new message cell M_r with receiver R, sender S, name, arguments, and a resend order of $r \geqslant 1$ is created. Normal messages use r=1, resent messages use r_{n+1}=r_n + 1.
3. The kernel calls `full_lookup` on the receiver R with message M_r as parameter. This creates a new lookup visitor V_n where count n=1, which is accepted by the receiver.
4. The lookup visitor V_n iterates over the delegation tree of the receiver, that is following delegations pointers. For branching cells, the branch pointer is followed first and then the delegation pointer. On each cell, `lookup` is called with M_r as parameter. If the visitor has not stopped, it continues.
5. If a cell C responds to the message M, the index n of V_n is compared to order r of M_r. If they are equal, the visitor stops and returns with the callable K provided by the lookup function of C. Otherwise, the index n of V_n is increased by one, and lookup continues.
6. If lookup was successful, the callable K is executed with message M_r as parameter, the returned value is returned to the sender S.

Why response counting rather than doing resends by simply continue lookup at the successor of the implementing cell? To answer that, we must distinguish between two kind of cells, branching cells and other cells. Branching cells have two delegation pointers (branch and trunk), others cells have one delegation pointer only.

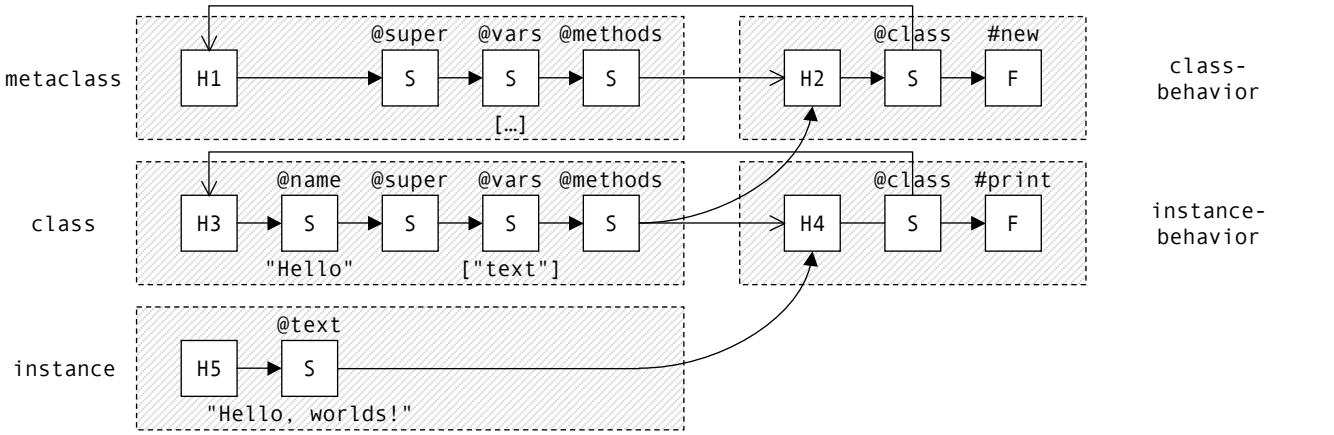


Figure 2. "Hello World" made of cells to illustrate the idiomatic structure of a class-based language realization with CELL: (from top left to bottom right) `H1` represents the metaclass, `H2` provides class behavior, `H3` represents the class, `H4` provides instance behavior, and eventually, `H5` represents an instance of the class.

Given a delegation chain without branchings, both resending approaches are equivalent. Given a delegation chain with a branching cell, that is a delegation tree, the approaches differ. A "successor resend" in the branch will never reach the cells in the trunk since the last cell of branch does not point back to the trunk. Response counting however restarts the (re)send from the receiver and thus takes both branch and trunk. As such, response counting is conceptually related to performing resends with a continuation of the original lookup.

3. Example 1: Hello World

This section provides an introductory example on how to realize a class-based language with CELL. In a class-based language there are three kinds of messages:

- Class messages can be sent without having an instance; they are understood by the class itself rather than the instance.
- Instance messages can only be sent if you have an instance; they are understood by each instance of the class.
- Instance-specific messages can only be sent to one instance; they are understood by one specific instance only.

This example exercises how to model a class that defines both class- and instance methods.

Figure 2 shows the structure of `Hello`, a class made of cells. There are two kind of cell groups: the groups on the left represent objects, and the groups on the right represent method dictionaries. Typically we denote groups by the name of their head cell. The cell groups are (from top left to bottom right):

- `H1` represents the metaclass of Hello. The main purpose of a metaclass is to store the class-method dictionary `H2` in its `#methods` slot.
- `H2` represents the class-methods of Hello. It is referred to by the `#methods` slot of the metaclass `H1`, and delegated to by the last cell of the class `H3`.
- `H3` represent the Hello class. Classes are instances of their metaclasses. The last cell of Hello delegates the class-method dictionary `H2`. Each class can create new instances of itself and stores the instance-method dictionary `H4` in its `#methods` slot.
- `H4` represents the instance-methods of Hello. It is referred to by the `#methods` slot of the class `H3`, and delegated to by the last cell of any instance, including `H5`.
- `H5` represents an instance of Hello. The instance group contains a slot for each instance variable, and the last cell typically delegates to the instance-method dictionary `H4`. As such, each instances has a separate state, while all share the same behavior.

  To install instance-*specific* behavior on an instance, one would insert additional function cells at the end of an instance group.

Actually printing Hello World works as follows (given in Python-like pseudo-code for better readability[3]).

```
# Prints "Hello, Worlds!"
hello = Hello.send("new")
hello.send("text=", "Hello, Worlds!")
hello.send("print")
```

The above structure distinguishes classes and behavior. Why? If the instance `H5` would simply delegate to its class `H3` rather than, as it is done now, to behavior `H4`, then, both class and instances would respond to the same messages—

[3] For better readability, any code examples in this paper are provided as Python code. Nevertheless, all examples presented in this paper are taken from the running examples of the current Java prototype, available under GPL license at `http://smallwiki.unibe.ch/cells`.

which is obviously *not* how class-based languages are supposed to work.

3.1 Creating New Instances of a Class

New instances of Hello are created by sending the message `#new` to the head cell `H3`. The Hello World example provided with the CELL prototype, implements instance creation as follows:

```
@bind(name = "new", arity = 0)
def create_instance(message):
    h3 = message.send("receiver")
    h5 = Head()
    for name in h3.send("vars"):
        h5.append(Slot(name))
    h4 = self.send("methods")
    head.append(h4)
    return h5
```

The above Python code does the following.

1. get the receiver of the message, that is, class Hello,
2. create the head cell of the to-be-created instance group,
3. get the list of instance variable names from Hello's `vars` slot,
4. for each name, append a new slot cell to the instance group,
5. get the instance-method dictionary, that is cell group `H4`, from Hello's `methods` slot,
6. append the dictionary to the the instance group, and
7. eventually, return the head of the instance group.

4. Cell Injection

Adding new cells to an existing group is either done by appending the new cell to the last cell of the group, or by inserting the new cell between two existing cells. We refer to this as *cell injection*.

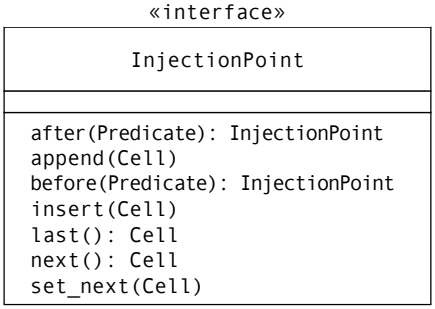


Figure 4. The InjectionPoint interface provides an API that can be used to find the correct "point of injection" for to-be-added cells.

The `InjectionPoint` interface provides an API that can be used to find the correct "point of injection" for to-be-added cells. Injection of new cells typically works in two steps as follows:

1. the *point of injection* is defined to be either before or after a cell that satisfies a given predicate,
2. the to-be-added cell is inserted at that point.

`before` injection points are typically used to inject cells that override or change existing behavior. Whereas `after` injection points are most often used to install cells with novel behavior just after the head cell of an object (or other unit of abstraction).

5. Example 2: Dwemthy's Array

This section illustrates how to extend a CELL-made high-level language using a `CustomLookup` cell. We considers Dwemthy's Array (13) as an example. Dwemthy's Array is a text based adventure game with a built-in coding challenge that uses the *method_missing* hook. Our implementation however will not use such a hook, but rather a CustomLookup cell.

In the game, a rabbit and creatures fight to death. One particular creature is Dwemthy's Array, an array filled with creatures that initially delegates all received messages to its first creature. If the creature dies, it is removed and thus the next creature appears. The array does not die until all its members are dead.

Figure 3 illustrates Dwemthy's Array made of cells: (from top left to bottom right) `H1` models Dwemthy's Array using a custom lookup cell, `H2` and `H4` are creatures contained in Dwemthy's Array, `H3` models the class behavior of the Creature class, `H5` models the rabbit, and eventually, `H6` models the instance behavior of the Creature class.

Creatures are created by sending `#new` to the cell `H3`. This creates a new head cell, appends four slot cells, initializes the slots, and eventually, links them to `H6` which provides the instance behavior of creatures.

The special behavior of Dwemthy's Array is implemented using a custom lookup cell that leverages lookup from the C2-level of the CELL kernel to the C3-level of the high-level language. Therefore, we can implement the lookup of Dwemthy's Array purely in terms of high-level sends:

```
def lookup(w_message):
    message = w_message.send("arg", 0)
    self = message.send("receiver")
    first = self.send("first")
    if first.send("life").value == 0:
        self.send("shift")
        if self.send("empty?").value:
            return Callable.constant(0)
        first = self.send("first")
    return first.full_lookup(message)
```

Please note, that by the protocol of `CustomLookup`, custom handlers receive the actual message as the first argument of a wrapper message named `#lookup`. Thus, our custom handler first unwraps the actual message, and then uses a series of high-level sends to dispatch the actual message on the first element of the array.

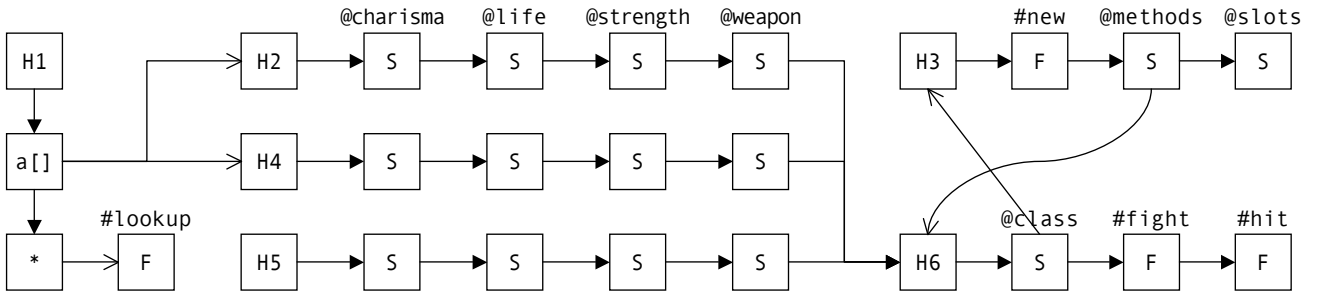


Figure 3. Dwemthy's Array made of Cells: (from top left to bottom right) `H1` models Dwemthy's Array using a custom lookup cell, `H2` and `H4` are creatures contained in Dwemthy's Array, `H3` models the class behavior of the Creature class, `H5` models the rabbit, and eventually, `H6` models the instance behavior of the Creature class.

6. Example 3: Smalltalk Made of Cells

This section provides a somewhat more complex example. A minimal Smalltalk system is made of cells, which is then considered by the two following sections as a case-study for introducing traits (19) and first-class object references (12).

"Smalltalk made of Cells" (or short CELLTALK) consists of the following four classes: `Object`, `Behavior`, `Class`, and `Metaclass`, each with an corresponding metaclass. Behavior inherits from Object, both Class and Metaclass inherit from Behavior. All four classes are the sole instances of their corresponding metaclasses. The inheritance of metaclasses parallels that of classes, with the addition that Object's metaclass inherits from Class. All metaclasses are an instance of Metaclass.

If you are unfamiliar with this setup, or even with Smalltalk's object model in general, please refer to Chapter 5 of the "Squeak by Example" book (4) for an introduction to the subject.

6.1 The Class Point and an Instance of Point

Figure 5 illustrates an (almost complete) cell graph of the CELLTALK kernel together with a Point class and a Point instance. On the left, the high-level abstractions of Smalltalk, *i.e.,* classes and objects, are shown as a UML class diagram, whereas the main part of the figure is taken up by cells that actually realizes these abstractions.

The Smalltalk abstractions are (from top to bottom):

- Everything is an `Object`. The classes of both classes and object ultimately inherit from Object. Of particular interest is the custom lookup handler at the end of Object's method dictionary. This cell implements the method-missing hook of Smalltalk-80 as follows:

```
def lookup(w_message):
    dnu = "doesNotUnderstand:"
    message = w_message.send("arg", 0)
    self = message.send("receiver")
    return self.send(dnu, message)
```

  In Smalltalk-80 and many other dynamic languages, hooks such as *method_missing* or *method_added* are popular extension points for custom language extensions. In CELLTALK however, this kind of hook is rendered obsolete as CELL's decomposition of objects into smaller parts offers better modularization and separation of concerns.

- `Behavior` is the common superclass of classes and metaclasses. Behavior implements `#new`, so instances of Behavior can create new instances of themselves. Instances of behavior have no state. Thus, `#new` is implemented as follows.

```
def new(message):
    self = message.send("receiver")
    method_dictionary = self.send("methods")
    head = Head()
    head.append(method_dictionary)
    return head
```

- `Class` extends Behavior with a name and named instance variables. It overrides `#new` using a resend as follows.

```
def new(message):
    self = message.send("receiver")
    head = message.resend()
    for name in self.send("vars").values:
        head.insert(Slot(name))
    return head
```

- `Object_class` is the metaclass of Object. It implements class-behavior of Object, which is in this case empty.

- `Point_class` is the metaclass of Point. It implements class-behavior of points, that is the constructor `#x:y:`. The constructor takes two arguments and is implemented as follows.

```
def x_y_(message):
    self = message.send("receiver")
    point = self.send("new")
    point.send("x:", message.send("arg", 0))
    point.send("y:", message.send("arg", 1))
    return point
```

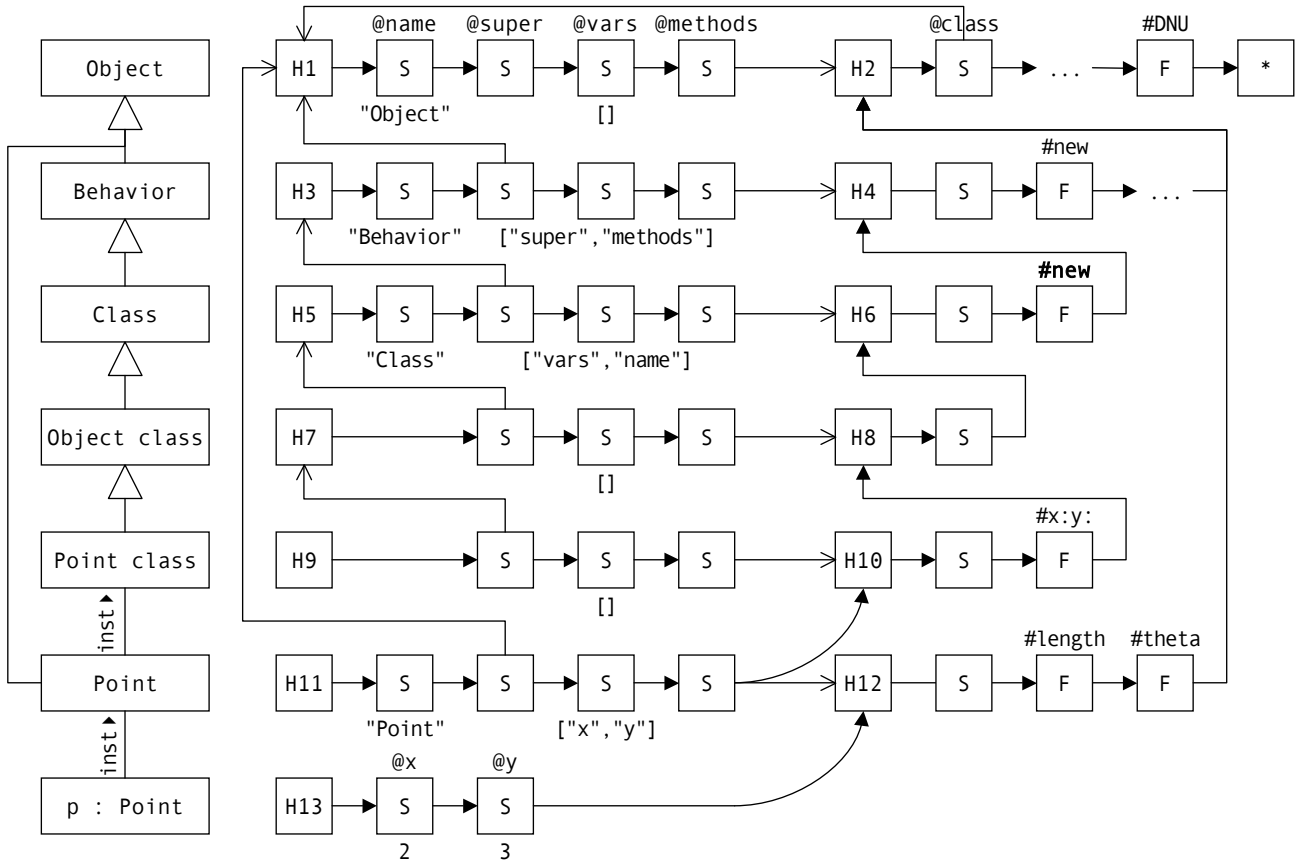


Figure 5. Smalltalk made of Cells: (from top to bottom), class Object, class Behavior, class Class, metaclass of Object, metaclass of Point, class Point, an instance of Point with instance variables x=2 and y=3. Cell groups in the center, that is with odd header numbers, represents objects. Cell groups to the right, that is with even header numbers, represent method dictionaries.

- `Point` is the class of Point. It specifies two instance variables named x and y, and implements instance-behavior of points. As an example, accessors for polar coordinates are shown on Figure 5.
- Eventually, the last item is an instance of Point. The instance is represented by two cell groups. First, `H13` represents the state of the instance, and second, `H12` is part of its class and provides instance-behavior. Thus each instance has its own state, whereas all instances share the same behavior.

However, in contrast to Smalltalk-80, in CELLTALK adding instance-behavior to an instance is straightforward. Just inject a new function cell either after `H13` or before `H12`!

Messages sent to the instance are looked up in `H13`, `H12`, and `H2` in this order. By contrast, lookup of class-methods (such as its constructor) starts at `H11`, and then traverses `H10`, `H8`, `H6`, `H4`, and `H2` in this order.

7. Example 4: Adding Traits

In this section, we extend the above CELLTALK kernel with first-class support for traits. Traits are composable units of behavior, similar to mixins (19), but avoiding fragility problems that may arise when modifying classes or mixins. When a class uses a trait, the trait contributes a set of additional methods that are based on one (or more) existing methods of the class. For example, the `TBound` traits contributes the methods `#bottom`, `#left`, `#right`, `#top`, and requires the method `#bounds`. Traits have no state and can inherit from one or more other traits.

Semantically, traits can be flattened away (15). In practice, this means that traits can be introduced experimentally to an existing class-based language "on the cheap" by flattening them away to the host language (14). This may be done because typical implementations of class-based languages do not allow one to extend the language with other unit of abstractions, such as traits. In CELL however, we can.

Figure 6 illustrates how we extend the CELLTALK kernel with traits. On the left, the high-level abstractions of Smalltalk, *i.e.,* classes, traits and objects, are shown in a UML class diagram, whereas the main part of the figure is taken up by cell-instances that actually realize these abstractions.

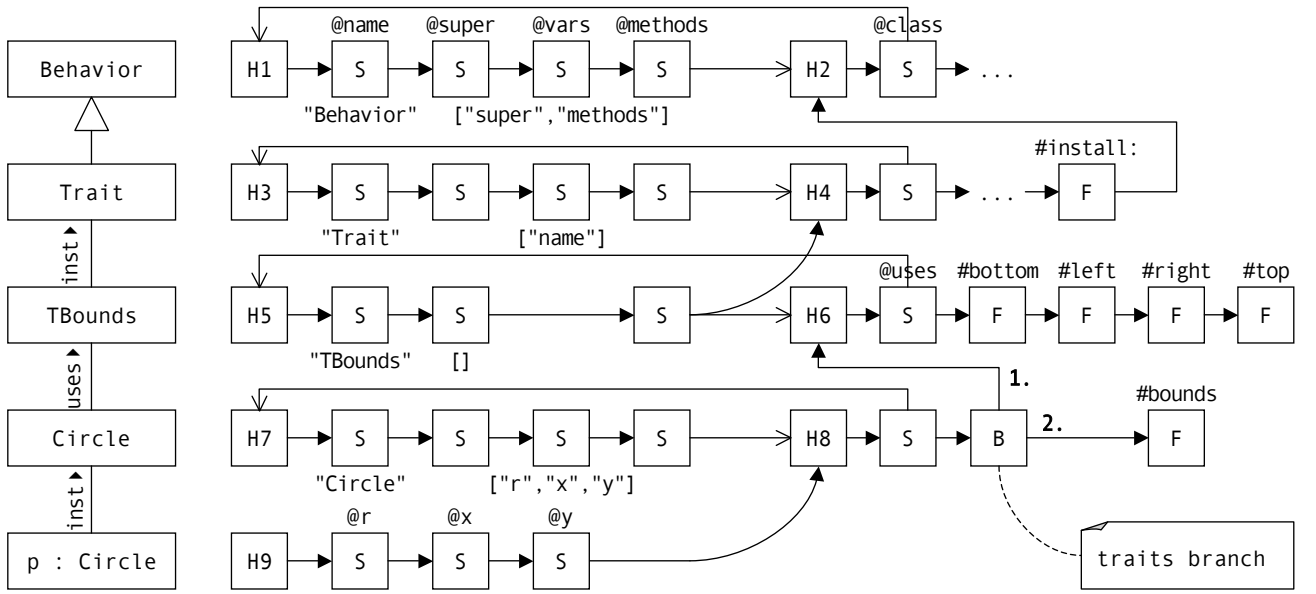


Figure 6. CELLTALK with Traits: (from top to bottom) class Behavior, class Trait, TBounds trait, class Circle that uses TBounds trait, an instance of Circle. The branching in the bottom right installs trait TBounds in the Circle class: messages are first sent to the methods provided by trait TBounds, and second only to the instance-methods provided by class Circle.

The Smalltalk abstractions are (from top to bottom):

- `Behavior` is the common superclass of Class and Metaclass. Instances of Behavior consist of a method dictionary and an inheritance relationship. Behavior is thus the natural extension point to add traits to the kernel.
- `Trait` extends Behavior with a name and functionally to allow classes to use traits using the following method:

```
def install_on(message):
    trait = message.send("receiver")
    class = message.send("arg", 0)
    m_dict = class.send("methods")
    branch = Branch.new(trait)
    m_dict.insert(branch)
```

  This inserts a branching into the method dictionary of the class. Branchings are cells with two delegation pointers: lookup is first delegated to the branch, then to the next cell.
- `TBounds` inherits from Trait. It provides four methods that all depend on the class's `#bounds` method.
- `Circle` is a class that uses the trait `TBounds`. Circle itself provides an `#bounds`.
- Finally, we have an instance of Circle.

When `#top` is sent to the Circle instance, lookup reaches the branching and takes the first branch. In the branch, the message is matched and thus the `#top` method of the trait is executed with the circle instance as receiver. In the body of this method, `#bounds` is called. Again, lookup starts at the circle instance, reaches the branching and takes the first branch. But none of the trait methods match the message, thus, lookup continues within the method dictionary of the class Circle, where the message is eventually matched by the `#bounds` method.

Neither the trait `TBounds` nor the provided methods contain any reference whatsoever to the class Circle or the circle instance. Therefore, the same trait can be used by any number of classes and any number of instances.

7.1 On Resend and Super-send Semantics

It is in the interaction between modules to support traits and class-inheritance that super-send semantics becomes interesting. In the presented traits example, the resend semantics of CELL are used to realize super-sends. Thus, if a trait-method does a super-send, the lookup visitor first visits all super-traits of the current class, and then the super-classes. However, this is not in accordance with the conventional flattening property of traits (15). Since traits are flattened, any super-send from a trait-method should behave as if sent from the class that imports the trait.

It remains open for further investigation how the desired behavior of flattened traits could best be realized at the intermediate level. Even though we introduced response counting to route resent message from tree branches back to the trunk, the resend semantics for CELL are limited. It is most likely best to distinguish between "resend" as a concept of the intermediate layer and "super-send" as a concept of the high-level language, rather than trying to directly map super-sends from the higher-level to resends at the lower-level.

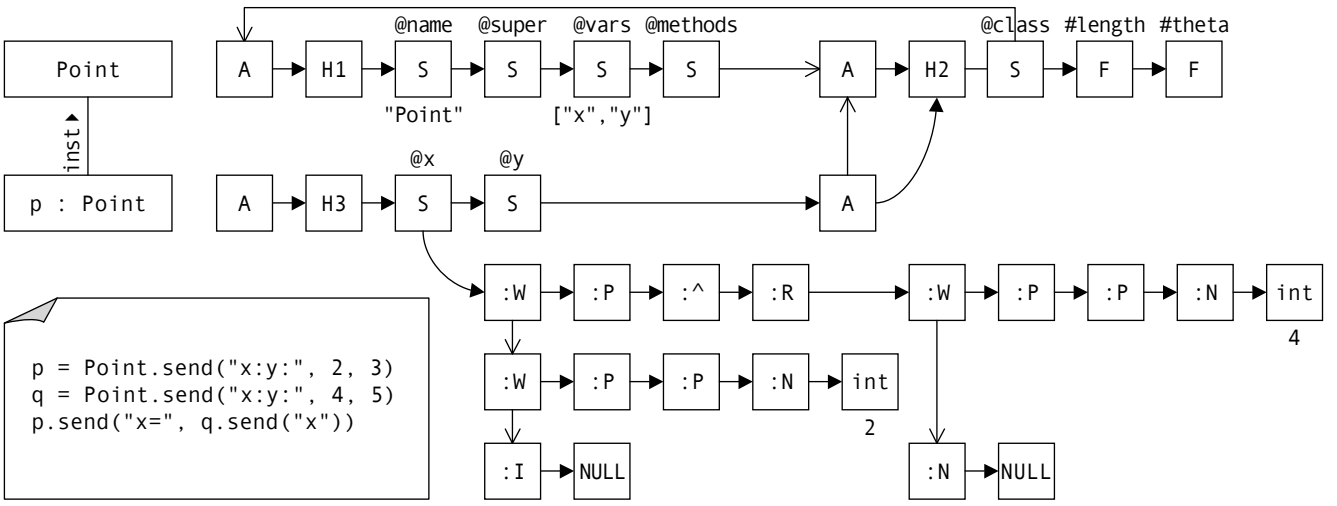


Figure 7. CELLTALK with Aliases: the complete alias tree of one slot cell is shown, together with the sequence of statements that led to the creation of these aliases. The following notation is used: write aliases are denoted as `:W`, read aliases as `:R`, parameter aliases as `:P`, return aliases as `:E`, allocation as `:A` and initialization as `:I`.

8. Example 5: Adding First-Class References

In this section, we extend CELLTALK with first-class object reference. Conceptually, we use the same approach as Adrian Lienhard's OBJECTFLOWVM (12), which extends the virtual machine of Smalltalk-80 with first-class object references.

Lienhard uses aliases as an abstraction to capture how object references propagate through a running system. All objects in the system are indirectly references through an alias. Each time an object is passed along in the running system, a new alias is created. The new alias references the same object as the originating alias, and additionally, maintains a pointer to the originating alias.

Different types of aliases are used to represent the different sources of new object references:

- `Allocation` of an object,
- `Initialization` of a slot,
- `Parameter` passing into a method,
- `Read` access to a slot,
- `Return`ing values from a method,
- and `Write` access to a slot.

Write aliases maintain two pointers: one to the originating alias, and another to the alias that was previously contained in the accessed slot.

Lienhard implements aliases as objects of the high-level language that are transparently added by the VM whenever an object reference is passed along. Our implementation however is achieved purely at the level of cell composition, so no changes to the underlying CELL kernel or virtual machine are required. Extending CELLTALK with aliases is simply a matter of changing the high-level methods that create or manipulate high-level instances.

- `Behavior#new` is changed so that it wraps the head of the just created instance group in an Allocation alias.
- `Behavior#putMethod:` is changed so that all installed functions are wrapped to create Parameter and Return aliases whenever they are called.
- `Class#new` is changed so that all created slots are wrapped into a function that creates Read and Write aliases upon access to the slot. Furthermore, all slots are initialized with an Initialize alias.

Figure 7 illustrates an (incomplete) snapshot of a running CELLTALK system with aliases. In the upper part, class Point and an instance of Point are shown. In the lower part, the complete alias tree of cell slot $p.x$ is shown.

In contrast to plain CELLTALK (see Figure 5) the cell groups that represent class Point and its instance are not referenced by their head cells but rather through alias cells. Nevertheless, head cells are still used as the identity of these groups. As a consequence, alias cells respond to the message `#==`, so identity comparison of alias cells can be delegated to the referenced head cells.

The lower part of Figure 7 shows the alias tree stored in the `#x` slot of the Point instance `p`. The sequence of statements that led to this alias cells is as follows (given in Smalltalk syntax):

```
p := Point x: 2 y: 3
q := Point x: 4 y: 5
p.x: q.x
```

Or, as expressed in terms of CELL message sends:

```
p = Point.send("x:y:", 2, 3)
q = Point.send("x:y:", 4, 5)
p.send("x=", q.send("x"))
```

Let's for example follow the flow of the integer 4 on Figure 7: it has first been initialized, then passed to the constructor of q, then passed to the setter of $q.x$ and written to the x slot of q. Later, it has been read from the slot and returned from a call to the reader of $q.x$. This return value has been passed to the setter of $p.x$, and eventually been stored in the x slots of p.

9. Discussion

The current implementation of CELL has been deliberately limited to first getting the concepts right. The semantics of CELL are currently given as Python code only; in the future we would like to provide a operational semantics for the language, potentially as a calculus of evolving objects (7).

The intermediate model presented in Section 2 provides language designers with 10 different types of cell abstractions. However, this set is not necessarily complete. In a language that supports more built-in data types besides integers and strings, the number of different cell types might quickly expand. It remains thus open to further investigation how to best implement the differing capabilities of the different cell types. There is a trade-off between complexity and extensibility. If the cell types are provided as built-in primitives of the intermediate language the set of different cell types will not be extensible. On the other hand, if there is a general abstraction that supports different types of cells the intermediate model will be directly extensible at the cost of an additional layer of complexity.

A practical limitation of CELL is runtime performance. Performing lookup on long delegations chains of cells typically results in a linear search over all understood message names. Therefore, we plan to introduce two new basic cell types, namely `Function`- and `SlotDictionary`. Consecutive runs of either Function or Slot cells can be replaced with a such a dictionary. However, for the sake of cell injection, this optimization must remain transparant to injection-clients. For example, injecting a head cell in the middle of a long run of optimized slots cells must split the according SlotDictionary into two SlotDictionaries separated by the injected head cell. More such optimizations are possible as long as they remain transparent to lookup- and injection-operations. For example, message lookup performance can be further improved by installing caches on all, or at least some cells. Head cells are particular candidates for caches, as most often lookup starts with head cells. However, as lookup in CELL is deliberately not limited to mere message name matching, not all lookup results can be caches. Lookup results that depend on contextual information that might change between two subsequent calls of the same message are not to be cached or else bad things may happen. For example, imagine a lookup that takes context information such as user authentication into account. It is clear that, in this case, caching lookup responses beyond a user session leads to a security leak.

The present prototype of CELL is further limited by not providing a complete interpreter environment. Currently, both call stack and garbage collection of the implementing language are used. The same applies for syntax and byte code. All this is done without loss of generality, as obviously Cells can be extended to provide such functionality on their own. One possible direction for further development of the current prototype is to port its sources to RPython, in order to use the PyPy tool chain (18) to automatically generate full-fledged Cells VMs for different back-ends.

10. Related Work

The use of object fragmentation in CELL is obviously reminiscent of Ossher's "sea of fragments" proposal (16). Ossher proposes to represent objects as collaborating groups of fragments with delegate-relationships between them. Each fragment contributes a part of the object's functionality. The key motivation is to provide a better representation of aspects of the intermediate language of virtual machines. The proposal neither provides a running prototype nor does it go into the details of implementing such a system.

The language model of CELL has predecessors in the Aspect Machine of Haupt and Schippers (8) and the ObjectFlow VM of Adrian Lienhard *et al.* (12). Both approaches split objects into at least two fragments, a head fragment and the actual object.

In the Aspect Machine of Haupt and Schippers (8) each object is represented by two or more fragments: a head fragment, zero or more proxy fragments, and a body with the entire object. The proxy fragments are used to install aspects in the running system. The authors propose to build a virtual machine based on that model, with common byte-codes for object manipulation and dedicated byte-code set to support proxy manipulation. One key difference is that the CELL model does not distinguish between fragments and actual objects. As a consequence the same set of operations (or byte-codes) can be used to manipulate both objects and aspects.

In the ObjectFlow VM of Lienhard *et al.* (12) each object is represented by two or more fragments: one or more head fragments (*i.e.,* aliases) and a body with the actual object. Section 8 presents a CELL implementation of their approach. The head fragments are introduced to keep track of object references. The authors use their system to realize a back-in-time debugger.

Composing classes from smaller fragments is not novel. Composition filters similarly enhance class abstraction in a modular way by decomposing into smaller fragments (1). In the model of CELL, objects are decomposed into fragments as well, and furthermore, both classes and objects are decomposed using the same set of fragments.

Also related is Daniel Bardou's work on split objects (3). The author uses head fragments to represent different roles and viewpoints of the same object. All head fragments associated with, *i.e.,* referring to, the same object have the same identity. This is analogous to what Cell does with aliases, since all aliases pointing to the same head cell have the same identity.

11. Concluding Remarks

In this paper, we have presented an intermediate model for representing object-oriented languages. The model breaks classes and objects into smaller fragments referred to as *cells*. A basic set of 10 cell types is given. Composition is used to realize various forms of method lookup and modularization mechanisms.

Since UML has not been designed with cells in mind, a novel graphical notation for the depiction of cells is introduced and used throughout the paper.

A running prototype of CELL is provided, together with examples that demonstrate the following capabilities of the model:

- CELLTALK a basic Smalltalk kernel is introduced in Section 6. We show how to model a class-based language made of cell composition. In particular, the separation of classes and behavior is identified as being idiomatic for such languages.
- Custom lookup is presented in Section 5 as a replacement of the *method_missing* hook often found in dynamic languages.
- Traits are added to CELLTALK to demonstrate the usefulness of cell composition for implementing novel modularization mechanisms (in Section 7).
- Aliases are added to CELLTALK to demonstrate the usefulness of cell composition for back-in-time debugging (in Section 8).

A mechanism for re-sending of message is presented. However, the traits example reveals that a general approach for realizing super-sends remains in need of further investigation.

The present prototype has been deliberately limited on first getting the concepts right, before optimizing for performance. It is however our conviction, that — now that the concepts are stabilizing — CELL can be tuned to become a performant and powerful intermediate machine, that can be used for realistic implementation of high-level languages with novel modularization mechanisms.

Acknowledgments

We gratefully acknowledge the financial support of the Swiss National Science Foundation for the projects "Analyzing, capturing and taming software change" (SNF Project No. 200020-113342, Oct. 2006 - Sept. 2008) and "Bringing Models Closer to Code" (SNF Project No. 200020-121594, Oct. 2008 - Sept. 2010).

The authors thanks Adrian Lienhard for his help with the implementation of aliases, and Michael Haupt and Hans Schippers, who granted early access to their MDSOC implementation, from which parts of the CELL implementation are derived.

A. Pseudo-Code of CELL lookup

The following pseudo-code illustrates how lookup is implemented in CELL. Please refer to Subsection 2.1 for more information.

```
class Cell(object):
    def __init__(self):
        self.next = None

    "Accept lookup- or injection-client."
    def accept(self, client):
        client.visit(self);
        if client.stopped: return
        if self.next is None: return
        self.next.accept(client);

    "Lookup message on self and all delegates."
    def full_lookup(self, message):
        lookup = Lookup(message)
        self.accept(lookup)
        if not lookup.stopped: return None
        return lookup.fun

    "Check if self (excluding delegates) responds."
    def implements(self, message):
        return self.lookup(message) is not None

    "Lookup message on self (excluding delegates).
    Must reply with a callable or None."
    def lookup(self, message):
        raise "Subclass responsability"

    "Convenience method for message sending."
    def send(self, name, args):
        w_args = [System.wrap(x) for x in args]
        message = Message(self,1,name,w_args)

    "Lookup message and execute the reply."
    def send(self, message):
        fun = self.full_lookup(message)
        if fun is None: raise Exception
        return fun.call(message)

    "Check if self or any delegate responds."
    def responds_to(self, message):
        return self.full_lookup(message) is not None

class Callable(Cell):
    def __init__(lambda):
        self.lambda = lambda

    "Execute this callable."
    def call(self, message):
        return self.lambda(message)

class Message(Cell):
    def __init__(self, R, order, name, args):
        Cell.__init__(self)
```

```
        self.receiver = R
        self.order = order
        self.name = name
        self.args = args

    "Resend this message."
    def resend(self):
        m = Message(
            self.receiver,
            self.order + 1,
            self.name,
            self.args)
        return self.receiver.send(m)

class Lookup(object):
    def __init__(self, message):
        self.message = message
        self.count = message.order
        self.stopped = False

    "Refer to Section 2.1 for more details."
    def visit(self, cell):
        self.fun = cell.lookup(self.message)
        if self.fun is not None:
            if self.count == 1:
                self.stopped = True
            else:
                self.count = self.count - 1
```

References

[1] Mehmet Aksit, Ken Wakita, Jan Bosch, Lodewijk Bergmans, and Akinori Yonezawa. Abstracting object interactions using composition filters. In Rachid Guerraoui, Oscar Nierstrasz, and Michel Riveill, editors, *Proceedings of the ECOOP '93 Workshop on Object-Based Distributed Programming*, volume 791 of *LNCS*, pages 152–184. Springer-Verlag, 1994.

[2] Michael Bächle and Paul Kirchberg. Ruby on Rails. *IEEE Software*, 24(6):105–108, 2007.

[3] Daniel Bardou and Christophe Dony. Split objects: a disciplined use of delegation within objects. In *Proceedings of OOPSLA '96*, pages 122–137, October 1996.

[4] Andrew Black, Stéphane Ducasse, Oscar Nierstrasz, Damien Pollet, Damien Cassou, and Marcus Denker. *Squeak by Example*. Square Bracket Associates, 2007. http://SqueakByExample.org/.

[5] Christoph Bockisch and Mira Mezini. A flexible architecture for pointcut-advice language implementations. In *VMIL '07: Proceedings of the 1st workshop on Virtual machines and intermediate languages for emerging modularization mechanisms*, page 1, New York, NY, USA, 2007. ACM.

[6] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: adding genericity to the Java programming language. In *Proceedings OOPSLA '98, ACM SIGPLAN Notices*, pages 183–200. ACM Press, 1998.

[7] Mariangiola Dezani-Ciancaglini, Paola Giannini, and Oscar Nierstrasz. A calculus of evolving objects. In *Proceedings of the 6th International Workshop on Multiparadigm Programming with Object-Oriented Languages (MPOOL 2008)*, 2008.

[8] Michael Haupt and Hans Schippers. A machine model for aspect-oriented programming. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP'07)*, volume 4609 of *LNCS*, pages 501–524. Springer Verlag, 2007.

[9] Erik Hilsdale and Jim Hugunin. Advice weaving in AspectJ. In *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 26–35, New York, NY, USA, 2004. ACM.

[10] Adrian Kuhn. Collective behavior. In *Proceedings of 3rd ECOOP Workshop on Dynamic Languages and Applications (DYLA 2007)*, August 2007.

[11] Bill Lewis. Debugging backwards in time. In *Proceedings of the Fifth International Workshop on Automated Debugging (AADEBUG'03)*, October 2003.

[12] Adrian Lienhard, Tudor Gîrba, and Oscar Nierstrasz. Practical object-oriented back-in-time debugging. In *Proceedings of the 22nd European Conference on Object-Oriented Programming (ECOOP'08)*, volume 5142 of *LNCS*, pages 592–615. Springer, 2008. ECOOP distinguished paper award.

[13] Why Malsky. *Why's (poignant) Guide to Ruby*. Creative Commons, 2005. http://www.poignantguide.net.

[14] Oscar Nierstrasz, Stéphane Ducasse, Stefan Reichhart, and Nathanael Schärli. Adding Traits to (statically typed) languages. Technical Report IAM-05-006, Institut für Informatik, Universität Bern, Switzerland, December 2005.

[15] Oscar Nierstrasz, Stéphane Ducasse, and Nathanael Schärli. Flattening Traits. *Journal of Object Technology*, 5(4):129–148, May 2006.

[16] Harold Ossher. A direction for research on virtual machine support for concern composition. In *VMIL '07: Proceedings of the 1st workshop on Virtual machines and intermediate languages for emerging modularization mechanisms*, page 5, New York, NY, USA, 2007. ACM.

[17] Frédéric Pluquet, Stefan Langerman, Antoine Marot, and Roel Wuyts. Implementing partial persistence in object-oriented languages. In *Proceedings of the Workshop on Algorithm Engineering and Experiments, ALENEX 2008, San Francisco, California, USA, January 19, 2008*. ACM-SIAM, 2008.

[18] Armin Rigo and Samuele Pedroni. PyPy's approach to virtual machine construction. In *Proceedings of the 2006 conference on Dynamic languages symposium, OOPSLA '06: Companion to the 21st ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 944–953, New York, NY, USA, 2006. ACM.

[19] Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew Black. Traits: Composable units of behavior. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP'03)*, volume 2743 of *LNCS*, pages 248–274. Springer Verlag, July 2003.

[20] Dave Thomas. Message oriented programming. *Journal of Object Technology*, 3(5):7–12, May 2004.

[21] David Ungar and Randall B. Smith. Self: The power of simplicity. In *Proceedings OOPSLA '87, ACM SIGPLAN Notices*, volume 22, pages 227–242, December 1987.

Predicate dispatch for Aspect-Oriented Programming

(Position paper)

Shigeru Chiba

Tokyo Institute of Technology
2-12-1 Ohkayama, Meguro-ku,
Tokyo 152-8552, Japan
www.csg.is.titech.ac.jp/~chiba

Abstract

Developing a machine model natively supporting aspect-oriented programming (AOP) is fruitful not only for implementing interpreters and compilers for AOP languages but also for understanding the essence of the AOP paradigm. This position paper shows a machine model based on open classes and predicate dispatch and it briefly describes a list of predicates that are necessary for modeling AspectJ. This machine model is useful for comparing AOP and object-oriented programming (OOP) on a side-by-side basis. Our initial observation is that AOP is a natural extension to OOP with respect to language constructs.

1. Introduction

Early aspect-oriented programming (AOP) languages such as AspectJ were implemented as a program translator into a program (or machine code) written in a non-AOP language. These days, several virtual machines that directly support AOP such as Steamloom [2] have been developed but their support is still limited [5]. This is partly because of the lack of a good machine model with native support for AOP.

To develop such a machine model for AOP, Haupt and Schippers proposed the concept of virtual join points [5] and showed the delegation-based AOP model based on that concept. In their paper, they presented similarity between join points in AOP and method calls in object-oriented programming (OOP) and AOP makes join points *virtual* as OOP makes function calls virtual (if we follow the C++ terminology). Here, join points are execution points in which two pieces of code are connected statically or slightly dynamically by the dynamic method dispatch of OOP. AOP gives another kind of late-binding feature to the join points. The delegation-based AOP model implements this virtualization by inserting a proxy object into a message delegation chain. In this model, all objects are prototype based and a method dispatching mechanism is represented by message delegation. Although a proxy can implement various kinds of advice, this model requires the delegation-chain mechanism to be extended for supporting a new pointcut primitive. For example, as shown in their paper, to support the cflow pointcut, they had to introduce a new kind of delegation chain that is effective only for a particular thread. Since the delegation chain is a fundamental component of the prototype-based object system, it should not be modified to support a new pointcut primitive.

This paper presents another machine model for AOP. It is still based on the concept of virtual join points but it uses an extended version of predicate dispatch [4, 6] as the basic mechanism. We believe that predicate dispatch is more intuitive for implementing the virtual join points than the delegation chain. Presenting similarity between predicate dispatch and the pointcut-advice of AOP is not new. This fact has been pointed out in a few papers [7, 1, 5]. The contribution of this paper is to show what predicates are really necessary for emulating pointcuts provided by AspectJ. We believe that our model is also useful to understand AOP by side-by-side comparison with OOP because predicate dispatch is a natural extension to the dynamic method dispatch of OOP. It would be also possible to introduce good ideas, such as the ambiguity property, for method dispatching in OOP into AOP.

2. Basic constructs

In our model, objects are instances of classes, which can consist of multiple modules as in open classes [3] and Hyper/J [8]. Suppose that we have the following class:

```
class Shape {
  Position p;
  Position getPos() { return p; }
  void setPos(Position np) { p = np; }
```


VMIL'08, October 19, 2008, Nashville, TN.

```
}
```

Then we can define a complementary module that adds some members to the Shape class. It would look something like this:

```
class Updater refines Shape {
  Display d;
  void setDisplay(Display nd) { d = nd; }
  void setPos(Position np) {
    d.repaint();
    // call a less-specific method with np.
  }
}
```

This Updater class corresponds to an aspect in the sense of AspectJ. We can define any number of classes that refine the Shape class. Updater itself is not a complete class; it cannot be instantiated. It is the name of a part of the declaration of the Shape class.

The Updater class adds a field d and a method setDisplay. These correspond to intertype declaration. Thus, at runtime, the Shape class has not only p, getPos, and setPos but also the added members d and setDisplay. An instance of the original Shape class includes those added members even if it is created by the code that does not expect the existence of Updater at all.

Note that the Updater also adds a method setPos, which is also declared in Shape. It is an advice. In this model, advising a join point is represented by overriding a method. The setPos method in Updater overrides the setPos method in Shape as an around advice in AspectJ does. Although we use the term *override*, the semantics of our overriding is slightly different from the normal one. The two implementations of setPos method given by Updater and Shape belong to the Shape class at runtime. However, since the implementation given by Updater *overrides* the other, it is invoked when the setPos method is called on an instance of Shape. No "multiple methods are applicable" error will be thrown. This is the same as the behavior of an around advice with the execution pointcut, which is invoked instead of the method specified by that execution pointcut when it is called.

3. Predicate dispatch

Since AspectJ provides several pointcut designators, simple method overriding is not sufficient to model AspectJ. Thus, we introduce an extended version of predicate dispatch. In a language supporting predicate dispatch, such as JPred [6], a method declaration can include predicates and it is invoked only when all the predicates are true. The original predicate dispatch allows only predicates that take local contexts such as parameter values and the receiver object (*i.e.* the this variable in Java). For example, JPred provides a predicate that becomes true if a parameter value is an instance of a specific class. This restriction is necessary for modular type checking and compilation.

However, since AOP is a paradigm for dealing with a crosscutting concern, our model for AOP needs some predicates that deal with non-local contexts. In other words, enabling method dispatch depending on non-local contexts is a unique feature of AOP against OOP. For example, see the following aspect in AspectJ:

```
aspect Logging {
  around(): call(void HashMap.put(..))
            && within(WebApp)) {
    System.out.println(
            "WebApp updates a hash map");
    proceed();
  }
}
```

This typical logging aspect declares an advice that is invoked when any method in the WebApp class calls the put method on a HashMap object. If a method in other classes calls it, the advice is not invoked but the original put method is invoked.

In our model, this advice is interpreted as a method overriding the put method in the HashMap class only when the client object that calls put is an instance of WebApp. Hence we need a predicate that checks the type of the client object, which is part of non-local contexts. The aspect described by using our model would be something like this:

```
class Logging refines HashMap {
  void put(Object key, Object value)
          when client instanceof WebApp {
    System.out.println(
            "WebApp updates a hash map");
    proceed();
  }
}
```

The expression following when is the predicate for the put method. Here, client is a hidden parameter, which is available without explicit declaration. It refers to the client object that calls the method.

Our model accepts a method with some predicates. The method with predicates does not have to override another method that has no predicates. For example, a method m may have only the implementations with predicates; in certain runtime contexts, all the implementations for m might be ineffective and thus a "message not understood" error would occur because all their predicates are false. However, we can modify our model to fit AspectJ's semantics. In AspectJ, an advice always modifies the behavior of an existing method. To implement this semantics in our model, we must constrain all methods with predicates to override the *default* method, which has the same name and signature but no predicates.

The predicates available in our model is defined as the following:

<*predicates*> := <*predicate*>

| <*predicate*> <*op*> <*predicates*>

<*predicate*> := <*var*> instanceof <*type*>

| <*var*> statically-instanceof <*type*>

| <*var*> running <*method*>

| ! <*predicate*>

| (<*predicate*>)

<*var*> := this | client | client* | <*parameter*>

<*op*> := && | ||

As <*var*>, not only this (*i.e.* a callee object) and a call parameter but also client is available to indicate the client object that calls the method. Thus, instanceof predicates can represent the same conditions that this, target, args, and within pointcuts in AspectJ can do. statically-instanceof is for the call pointcut in AspectJ. It checks the static type of <*var*>, usually this. If the static type of the receiver object is <*type*> at the client site, this predicate:

this statically-instanceof <*type*>

becomes true. The static types of client and the parameters are always the same as their dynamic types. running is for the withincode and cflowbelow pointcuts. It becomes true if <*var*>, which is usually client or client*, is running the specified <*method*>. client* represents any object contained in the current call stack. Hence,

client* running <*method*>

becomes true only while the current thread of control is executing the specified <method>. This corresponds to the cflowbelow pointcut.

3.1 Exhaustiveness

The original predicate dispatch was carefully designed for the exhaustiveness property. This property guarantees that no "message not understood" error happens during runtime. For example, the compiler of JPred [6] can modularly check that this property is preserved. If predicate dispatch is naively available, it is not obvious whether or not a program has the exhaustiveness property because some methods may have implementations effective only under certain runtime conditions. If the compiler allows calling such a method, the call would cause a "message not understood" (*i.e.* the called method is not found) error when there is no implementation effective for the calling contexts.

Unfortunately, our model does not enable static exhaustiveness checks because a predicate may access client*, which is never statically determined. On the other hand, our modified model allows a compiler to statically check that a given program preserves the exhaustiveness property. Recall that we mentioned that, to make our model exactly fit AspectJ's semantics, we have to add an extra constraint to our model. Since this modified model constrains a method with predicates to override the default method, which does not have predicates, any method has an implementation that is always effective independently of runtime contexts. Thus, the compiler can statically check exhaustiveness if all the classes including *aspect* classes refining normal classes are known to the compiler. Since our model adopts open classes, the set of the methods available in a given class is not determined unless all the modules contributing to that class (a normal class and aspect classes refining it) are given.

To enable modular exhaustiveness checks, we must introduce another language construct. For example, as eJava [9] does, the client code must explicitly specify modules that declare methods used by that client code. If the client code uses the setDisplay method added by the aspect class view.Updater to the original class model.Shape, it must include the following statement:

use view.Updater;

Then the compiler can understand that the setDisplay method in the model.Shape class is declared in the source file of the view.Updater class, which refines model.Shape. This enables modular class-by-class compilation as the Java compiler does.

Note that the explicit import of modules via use statement is not necessary when client code invokes an overriding method given by an aspect class. The compiler can perform modular exhaustiveness checks if the called method is also declared in the original class or another aspect class explicitly imported via use. Suppose that the refining class view.Updater overrides the setPos method in the model.Shape class. The compiler can modularly perform exhaustiveness check on the client code that will call the setPos method whichever implementation will be invoked, view.Updater's or model.Shape's.

3.2 Ambiguity

Another interesting property is ambiguity. If a program is not ambiguous, a "multiple methods are applicable" error never occurs during runtime. Ambiguity checking ensures that there are no two methods that have the same name and signature and also have predicates that become true in the same runtime contexts.

Since AspectJ allows multiple advices modifying the behavior of the same join point, our model does not guarantee that all programs preserve the ambiguity property. Two aspect classes refining the same normal class may override the same method and their two overriding methods might be effective at the same time.

In AspectJ, programmers can explicitly specify the precedence order among aspects. We can introduce a similar mechanism into our model so that the ambiguity of method dispatch will be resolved. Then a compiler will be able to conservatively check that all necessary precedence order is given.

We use predicates for representing the precedence order among aspect classes. The most specific method among ef-

fective overriding methods is determined by using logical implication relations among predicates. Note that the original predicate dispatch also uses implication relations for determining the most specific method. If two methods m_1 and m_2 are included in the effective methods for a method call and m_1's predicate expression logically implies m_2's predicate expression, then m_1 is more specific method than m_2. For example, suppose that m_1's predicate expression is p1 instanceof Rect and m_2's predicate expression is p1 instanceof Shape, where p1 is the first parameter of m_1 and m_2 and Rect is a subclass of Shape. Then m_1 is more specific than m_2 because, if p1 is an instance of Rect, then it is always an instance of Shape.

We implement the precedence order by specializing this algorithm for determining the most specific method. Let us introduce a new predicate deploy. It takes one aspect class name as a parameter. For example, deploy(Logging) is true only when an aspect class named Logging is deployed. If another aspect class Updater is also deployed and Logging has higher precedence than Logging, then deploy(Logging) logically *implies* deploy(Updater). The parameter of deploy can be null. deploy(null) is logically implied by deploy(A) for any aspect class A. In summary,

deploy(*A*) logically implies deploy(null) for any *A*.

deploy(*A*) logically implies deploy(*B*)
if *A* has higher precedence than *B*.

In our model, any method implicitly has a deploy predicate. If the method is declared in an aspect class A , which refines another class, then it has deploy(A). Otherwise, if it is declared in a normal class, it has deploy(null). If two methods m_1 and m_2 are effective and m_1's deploy logically implies m_2's deploy, then m_1 is a more specific method than m_2 and hence m_1 overrides m_2. Other predicates are not used to determine which method is more specific although it is possible to enhance our model to use other predicates.

3.3 super **and** proceed

By introducing a deploy predicate, we could make our model show similar behavior as AspectJ. However, to make our model exactly emulate AspectJ's semantics, we need more.

AspectJ has an algorithm for determining which advice is first executed, that is, which method is the most specific. However, this algorithm is not modular. Suppose that a class Shape has a method setX. An aspect Logging declares an advice α for calls to setX. There is another class Rect, which is a subclass of Shape and overrides the method setX. Calls to the method setX in Rect is advised by an advice β declared in an aspect Updater. The aspect Logging has higher precedence than Updater.

If the setX method is called on an instance of Rect, then the advice α is first executed because Logging has the highest precedence. If α calls proceed, then the advice β is next executed. After that, the method setX in Rect is executed. If setX calls super.setX, then the advice α is executed again, and finally the setX in Shape is executed. The order of specificity for each method is α, β, Rect.setX, α (again), and Shape.setX. Note that the advice α is executed twice. Although α advises the setX method in the super class, it is executed for the call to the method in the subclass and it is executed again for the call on super.

To emulate this behavior, our model also has to support both super and proceed to invoke a less specific method:

- proceed(p1, p2, ...)
  This invokes a less specific method with parameters p1, p2, ... The order of specificity is determined by deploy predicates and declaring classes. This is usually used by methods in aspect classes.
- super.<*method*>(p1, p2, ...)
  This calls the same method on the same instance with using the super class as the dynamic type of that instance. This is usually used by methods in normal classes.

For the example above, the setX method in Rect should call super.setX(newX) in our model. The setX methods in the aspect classes Logging and Updating should call proceed(newX). Then the behavior of a call to setX on an instance of Rect is the same as in AspectJ.

4. Concluding remarks

This paper presents an AOP machine model based on open classes and predicate dispatch. This model leads us to regard AOP as a natural extension to OOP. The paper argued that the language constructs of AOP are fairly equivalent to an extended version of predicate dispatch and thereby it is possible to directly compare AOP with other OOP-based programming paradigms. The contribution of this paper is to show what predicates are really necessary for emulating pointcuts provided by AspectJ. The original predicate dispatch allows only predicates that take local contexts such as parameter values and the receiver object. This restriction is for modular type checking and compilation. On the other hand, to emulate pointcuts, we had to allow predicates that deal with non-local contexts such as the client object. This is necessary for modularizing crosscutting concerns. A drawback of this fact is that modular type checking and compilation is sacrificed.

The presented work is still at a very early stage and this paper shows only a very rough sketch of the machine model. A lot of work remains. Our model does not support pattern matching or several pointcuts such as if, handler, set, or get. It does not support aspect instances.

References

[1] Bockisch, C., M. Haupt, and M. Mezini, "Dynamic Virtual Join Point Dispatch." Workshop on Software Engineering Properties of Languages and Aspect Technologies (SPLAT '06), 2006.

[2] Bockisch, C., M. Haupt, M. Mezini, and K. Ostermann, "Virtual machine support for dynamic join points," in *Proc. of Int'l Conf. on Aspect-Oriented Software Development (AOSD 2004)*, pp. 83–92, 2004.

[3] Clifton, C., G. T. Leavens, C. Chambers, and T. Millstein, "MultiJava: modular open classes and symmetric multiple dispatch for Java," in *Proc. of ACM Conf. on Object-Oriented Programming Systems, Languages, and Applications*, pp. 130–145, ACM Press, 2000.

[4] Ernst, M., C. Kaplan, and C. Chambers, "Predicate Dispatching: A Unified Theory of Dispatch," in *ECCOP '98: Proc. of the 12th European Conference on Object-Oriented Programming*, pp. 186–211, Springer-Verlag, 1998.

[5] Haupt, M. and H. Schippers, "A Machine Model for Aspect-Oriented Programming," in *ECOOP 2007 – Object-Oriented Programming*, vol. 4609 of *LNCS*, pp. 501–524, 2007.

[6] Millstein, T., "Practical predicate dispatch," in *Proc. of ACM Conf. on Object-Oriented Programming Systems, Languages, and Applications*, pp. 345–364, ACM, 2004.

[7] Orleans, D., "Separating behavioral concerns with predicate dispatch, or, if statement considered harmful," in *Workshop on Advanced Separation of Concerns in Object-Oriented Systems at OOPSLA '01*, 2001.

[8] Ossher, H. and P. Tarr, "Hyper/J: multi-dimensional separation of concerns for Java," in *Proc. of the Int'l Conf. on Software Engineering (ICSE)*, pp. 734–737, 2000.

[9] Warth, A., M. Stanojević, and T. Millstein, "Statically scoped object adaptation with expanders," in *Proc. of ACM Conf. on Object-Oriented Programming Systems, Languages, and Applications*, pp. 37–56, 2006.

Aspects and Class-based Security

A Survey of Interactions between Advice Weaving and the Java 2 Security Model

Andreas Sewe Christoph Bockisch Mira Mezini

Technische Universität Darmstadt
Hochschulstr. 10, 64289 Darmstadt, Germany
{sewe, bockisch, mezini}@st.informatik.tu-darmstadt.de

Abstract

Various aspect-oriented languages, e.g., AspectJ, AspectWerkz, and JAsCo, have been proposed as extensions to one particular object-oriented base language, namely Java. But these extensions do not fully take the interactions with the Java 2 security model into account. In particular, the implementation technique of advice weaving gives rise to two security issues: the erroneous assignment of aspects to protection domains and the violation of namespace separation. Therefore, a comprehensive discussion of the design choices available with respect to interactions with the dynamic class loading facilities of the Java VM is provided.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Language Classifications—Multiparadigm languages, Object-oriented languages; D.4.6 [*Operating Systems*]: Security and Protection—Access controls

General Terms Languages, Security

Keywords Advice weaving, aspect-oriented programming, dynamic class loading, Java security model

1. Introduction

The paradigm of aspect-oriented programming (AOP) aims at the modularization of cross-cutting concerns [13], i.e., concerns which cut across modularizations as offered by other paradigms, e.g., across class hierarchies in the case of object-oriented programming (OOP). To this end, an *aspect language* typically extends a *base language* rooted in another paradigm with a new kind of module: the aspect. For various aspect-oriented languages, e.g., for AspectJ [12, 3], AspectWerkz [7], and JAsCo [18, 19], this base language was chosen to be Java [10], whose entire security model revolves around the sole kind of module known to it: the class. Thus, the question arises how to best integrate aspects with Java's class-centric model in the light of the Java VM's dynamic class loading facility. In this paper we argue for increasing the level of class loader awareness of execution environments geared towards AOP and show how failure to do so can have serious security implications. To bolster the argument we have surveyed the behavior of current implementations, characterized interactions between classes and aspects, and identified a desirable design.

The remainder of this paper is structured as follows. Section 2 provides background material on two subjects: The security model of Java is described in Section 2.1, whereas Section 2.2 describes aspect-oriented programming and the weaving technique used by its implementations. Section 3 lists the surveyed implementations, before Section 4 characterizes not only the class loading behavior their execution environments exhibit, but also the design we deem most desirable. Section 5 mentions further implementation issues, Section 6 cites related work, and Section 7 concludes our position and gives suggestions for future work.

2. Background

The interactions between aspects and the class-based security model of Java are often subtle. It is therefore crucial to understand both the design of Java's security model and common implementation techniques of aspect-oriented languages.

2.1 The Java 2 Security Model

The Java VM's facilities for dynamic class loading [14] are an intrinsic part of the Java 2 security model, colloquially called the "Java sandbox." Said model revolves around three core components: the `SecurityManager`, the `AccessController`, and the `ClassLoader`.

2.1.1 Access Control

All operations deemed critical in the core API of Java are subject to access control; permissions, e.g., to delete a file, are ultimately granted by a `SecurityManager`, as is shown in the following.

VMIL'08, October 19, 2008, Nashville, TN.

```
1  class File {
2    //...
3    boolean delete() {
4      SecurityManager m =
5        System.getSecurityManager();
6      Permission p =
7        new FilePermission(this.getPath(),
8          FILE_DELETE_ACTION);
9      if (m != null)
10       m.checkPermssion(p);  // May throw SecurityException
11     // Perform delete
12   }
13 }
```

With the advent of Java 2, this mechanism's flexibility has been greatly enhanced by the `AccessController` framework for user-defined policies [9]. Hereby the VM's `SecurityManager` bases its decision of whether to grant a permission both on a user-configurable policy and on the calling context of the request. To determine the latter, the `AccessController` inspects all so-called protection domains recorded on the call stack and grants only those permissions afforded by each of them.[1] Which domain a frame belongs to thereby depends on the declaring class of the frame's corresponding method; which domain the class in question belongs to is fixed during class loading, e.g., depending on the source the class's code has been obtained from.

2.1.2 Dynamic Class Loading

Class loaders have been introduced to the Java security model in order to facilitate multiple namespaces, a user-definable class loading policy, and type-safe linkage in the presence of lazy loading [14]. They are responsible for resolving symbolic references [15, §5.1], i.e., fully qualified class names, to `Class` instances. Furthermore, it is among the class loaders' responsibilities to assign any newly defined class to a protection domain. Since class loaders thus form one of the corner stones of Java's security model, both the creation of and access to class loaders are controlled by a `SecurityManager`.

The ability to create `ClassLoader` instances also entails the ability to define new namespaces which can cleanly separate trusted from untrusted classes. This is possible since a class's identity is not uniquely determined by its name alone; at least at run-time class loaders need also be taken into account. This is exemplified by Figure 1, which depicts a typical class loader hierarchy in which the class loader instances `AppletClassLoader@5` and `AppletClassLoader@6` are used to separate the Java applets defined by them not only from one another but also from other parts of the application; classes defined by the two instances are, e.g., unable to refer to classes defined by either `URLClassLoader` as they reside within different branches of the class loader hierarchy.

[1] Privileged actions, which exempt parts of the call stack from inspection, are beyond the scope of this paper (cf. Section 7).

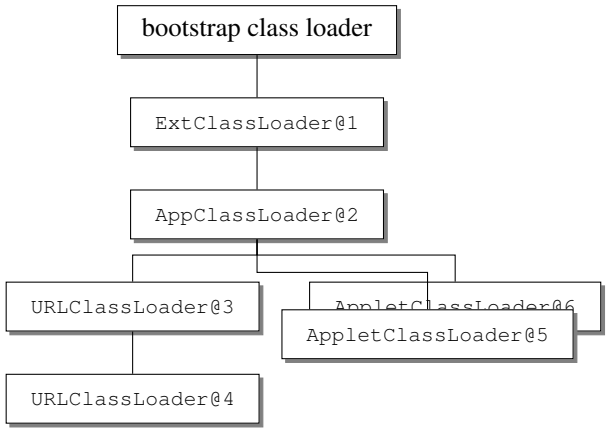


Figure 1: A class loader hierarchy, rooted at the bootstrap class loader [15, §5.3.1] and defining two separate namespaces for Java applets.

This behavior in which classes can refer only to classes defined by their own class loader or one of its ancestors in the hierarchy is only a convention—although one almost universally obeyed. What ultimately determines the `Class` returned upon a call to a class loader's `loadClass`[2] method is up to its (user-definable) implementation. Typically, this initiating class loader first delegates to its parent class loader before attempting to define the class on its own by means of its `defineClass` method. There are exceptions, though: Servlet containers follow the so-called delegation inversion model, which behaves the other way around—but can be emulated by the standard delegation model. But regardless of how delegation is handled, the Java VM places constraints upon loading which ensure type-safe linkage [15, §5.3.4]; thus, any delegation model is merely a set of guidelines on how to fulfill these constraints.

Even though these guidelines may appeal to our intuition of a class loader hierarchy, we cannot rely on any such behavior; thus, we have to adopt the following definitions [15, §5.3] and notation [14] to precisely describe class loading.

Definition 2.1. Let $c = l$`.loadClass("C")`. Then l is said to be the initiating class loader of the class c.

Definition 2.2. Let $c = l_c$`.defineClass("C")`. Then l_c is said to be the defining class loader of the class c, i.e., l_c defines c.

Note that due to delegation there may be multiple initiating class loaders of a class; however, for any class c there is always a single defining class loader l_c.

Notation 2.1. A class named "C" is denoted by $\langle \text{"C"}, l_c \rangle^l$, if l_c and l are its defining and initiating class loaders, re-

[2] All attempts to load a class dynamically are channeled through the `loadClass` method; the well-known `Class.forName`, e.g., simply defers class loading to this method.

spectively. If clear from context, this will be abbreviated to $\langle$"C", $l_c\rangle$ or $\langle$"C"$\rangle^l$.

2.2 Aspect-oriented Programming

In AOP's pointcut-and-advice flavor [16], which will be the focus of this paper, aspects affect the execution of so-called join points, e.g., calls to methods or accesses of fields. Hereby *pointcuts* select a set of join points at which—in addition to the action of the join point itself, i.e., the method call or field access—additional actions are to be performed in the form of so-called *advice*.

The following fragment exemplifies this; it defines a pointcut together with its associated advice which intercepts all calls to `delete` in order to subject this file I/O operation to access control as described in Section 2.1.1. Hereby the **`call`** atomic pointcut determines the *join point shadow*, i.e., the actual call instructions at which the advice will take effect, whereas the **`target`** atomic pointcut binds a context value, namely the callee. If permission is granted, the advice proceeds to the intercepted method, i.e., to `delete()`.

```
1  aspect AccessControl {
2    around(File f) : call(File.delete())
3       && target(f) {
4      SecurityManager m =
5        System.getSecurityManager();
6      Permission p =
7        new FilePermission(f.getPath(),
8          FILE_DELETE_ACTION);
9      if (m != null)
10       m.checkPermssion(p); // May throw SecurityException
11     proceed(f); // Proceed with f.delete()
12   }
13 }
```

The above illustrates the usefulness of the aspect-oriented paradigm to modularize cross-cutting concerns like access control, which would otherwise be tangled with code implementing totally unrelated concerns, e.g., performing file I/O.

In order to realize these semantics on top of the Java platform, implementations of aspect-oriented languages (cf. Section 3) typically "weave" a call to a synthetic advice method into the code of the base program [11], thereby altering classes other than the aspect; this is illustrated by the listing below, which completely replaces all calls to `delete()`.[3]

```
1 AccessControl a = AccessControl.aspectOf();
2 a.around$1(f, new AroundClosure$2());
```

Code like the above is generated at different times by different implementations, e.g., at compile-time, post-compile-time, load-time, or run-time. For the purpose of this discussion, the former two options have the same implications and will therefore be subsumed under the term static weaving. While they are altogether ignorant of dynamic class loading, it is impossible to weave into classes from a code source different from the aspect's. Since consequently only the security considerations for ordinary Java apply in this case, static weavers have been excluded from the survey.

Like compile-time and post-compile-time weavers, load-time and run-time weavers share a number of characteristics and will henceforth be subsumed under the term dynamic weaving. In contrast to run-time weaving load-time weaving has one caveat [2]: "All aspects to be used for weaving must be defined to the weaver before any types to be woven are loaded." This applies not only to AspectJ, whose Development Environment Guide this quote is taken from, but to other languages supporting load-time weaving as well, for otherwise classes may be "*missed* by [weaving] aspects added later, with the result that invariants across types fail."

3. Surveyed Implementations

We have surveyed the latest incarnations of several aspect-oriented languages all of which support dynamic weaving.

3.1 AspectJ 1.6

The AspectJ programming language [12] is arguably the most prominent aspect-oriented language, whose implementations can utilize not only compile-time and post-compile time weaving but also load-time weaving. Furthermore, two alternative implementations of the latter exist: One uses a Java 5 agent and one uses a custom class loader. Both implementations take dynamic class loading into account; it is thus claimed [2] that they "compl[y] with the Java 2 security model." Of these implementations the agent-based one has been our main object of study. Not only is it the most recent, but also not subject to further issues as described in Section 5; it applies a `class` file transformation upon class definition but does not otherwise interfere with dynamic class loading.

3.2 AspectWerkz 2.0

In contrast to AspectJ, AspectWerkz [7] is not so much an aspect-oriented language but a framework. Still, just as for AspectJ, class-loader aware implementations exist. In fact, AspectWerkz comes with a number of alternative implementations of run-time weaving. Several of these are specific to a single VM, e.g., to the JRockit or HotSpot VM [17]. One implementation, however, is universally available across all VMs supporting the Java 5 platform. Consequently, this agent-based implementation has been surveyed.

3.3 JAsCo 0.8.7

The design goal of the JAsCo language [18] was to combine aspect-oriented and component-based software development. As the latter typically makes—at least in a Java environment—heavy use of dynamic class loading, it stands to reason that JAsCo should have a high level of class loader awareness. Beyond mere static weaving JAsCo also offers two implementations supporting run-time weaving by means

[3] For readability, the listing is presented as Java source code, even though advice weaving is typically performed at the level of Java bytecode.

of HotSwap or a Java 5 agent. As the former implementation is marked deprecated, we have chosen the latter.

4. Characterization of Class Loading Behavior

The presence of protection domains and class loaders [9, 14] within the Java VM gives rise to two crucial questions: Which protection domain ought aspects be assigned to? And which classes ought to be affected by which aspects? While Section 4.1 answers the former question, Section 4.2 attempts an answer to the latter.

4.1 Protection Domain Assignment

As the protection domains are assigned by the class loader and all surveyed implementations compile aspects to ordinary `class` files, it is natural that an aspect gets assigned a protection domain based on the source of said file. But while this assignment seemingly integrates aspects with the Java security model, it is oftentimes jeopardized by the weaving technique used.

The reason for this is that optimizing weavers may decide to replace the call to the synthetic advice method (cf. Section 2.2) with the advice body itself. This implementation technique, known as inlining [4], severs the link of an aspect to its protection domain; the advice's code becomes part of the join point shadow's class. As such it belongs to the latter's protection domain in the eyes of the VM. This fact can be exploited by an aspect which is woven into a trusted class. Due to inlining the aspect's protection domain is no longer recorded on the call stack when the inlined advice's action is performed; consequently, all permissions granted to the trusted class are granted to the aspect as well. To prevent this escalation of privileges we therefore postulate that code contributed by an aspect should always be treated as belonging to that aspect's protection domain. This is feasible even in the presence of inling as virtual machines already cope with similar situations by maintaining a mapping from machine code addresses to methods [1, 17].

It should be noted that the above issue is independent of the issues surrounding **`privileged`** aspects [8],[4] which allow an aspect to call methods or access fields otherwise inaccessible to it. In contrast to access modifiers of method or fields, protection domains control, e.g., whether a file I/O operation may be performed; thus, they govern a different set of privileges.

4.2 Namespace Separation

In addition to the above issue, inlining may violate namespace separation: When resolving symbolic references an inlined advice would use the join point shadow's defining class loader instead of the class loader its declaring aspect is defined by. Similar to the issue of protection domain assignment all code contributed by an aspect should thus be assigned the aspect's defining class loader.

But irrespectively of the weaving technique used, there is the more general question of which aspects may affect which classes. When addressing this question, it is useful to abandon the notion of weaving (cf. Section 2.2) for the moment, and rather discuss design decisions in terms of virtual method calls alone,[5] as they ultimately give rise to the join points in question.

Consequently we focus on the three classes involved during method dispatch: the caller's dynamic type, the callee's static type, and the callee's dynamic type. Of these only the former two are decisive during resolution [15, §5.4.3], whereas the latter is not considered by the Java VM when resolving the symbolic reference. The callee's dynamic type is, however, useful when deciding whether a method call like `f.delete()` ought to be advised or not—at least, the existence of **`execution`** atomic pointcuts [5] in all surveyed languages stipulates this. Consequently, we take all three types and their class loaders into account. Hereby, b will denote the dynamic type of the caller in the base program, c will denote the static type of the callee, and d will denote the dynamic type of the callee. Their defining class loaders will be denoted by l_b, l_c, and l_d.

Utilizing the notation introduced in Section 2 we now define four cases which describe the relative position of two class loaders l_a, l_z in the class loader hierarchy—under the assumption that the standard delegation model is adhered to. If not, the definitions below approximate the class loaders' relationship as suggested by the names chosen. So, let $a = \langle\text{"A"}, l_a\rangle$ and $z = \langle\text{"Z"}, l_z\rangle$.

Same The equation $l_a = l_z$ captures the simplest case possible: Both classes are defined by the same class loader.

Ancestor-or-same This case, denoted by $l_a \succcurlyeq l_z$, is defined by the equation $a = \langle\text{"A"}\rangle^{l_z}$. Under the above assumption it reflects the intuition that the class loader l_a resides above l_z in the hierarchy. (**Ancestor**, denoted by $l_a \succ l_z$, is defined analogously.)

Descendant-or-same This case, denoted by $l_a \preccurlyeq l_z$, is defined by the equation $z = \langle\text{"Z"}\rangle^{l_a}$ and reflects the intuition of l_a residing below l_z in the hierarchy. (The analogously defined **Descendant** is denoted by $l_a \prec l_z$.)

Sibling If neither of the above three equations holds, l_a and l_z are assumed to reside in different branches of the hierarchy. This case is denoted by $l_a \lessgtr l_z$.

Note that neither $\succcurlyeq$ nor $\preccurlyeq$ are guaranteed to be transitive if the standard delegation model is not adhered to. If the above assumption holds, however, transitivity does hold as well. In the following we will indicate this by a distinct notation: $\geqslant$ and $\leqslant$.

[4] Privileged aspects are not to be confused with privileged actions.

[5] The discussion carries over to field accesses and static method calls.

	l_b	l_c	l_d	AspectJ AspectWerkz	JAsCo	Desired Design
l_a	$>$	$>$	$>$	✓	✓	✗
	$<$	$<$	$<$	✗	✗	✗
	$\not<$	$<$	$<$	✗	✗	✗
	$<$	$<$	$\not<$	✗	✗	✗
	$\not<$	$<$	$\not<$	✗	✗	✗
	$\not<$	$\not<$	$\not<$	✗	✗	✗
	$\geqslant$	$\leqslant$	$\geqslant$	✓	✓	✓
	$\geqslant$	$<$	$<$	✓`call`	✗	✓
	$\geqslant$	$<$	$\not<$	✓`call`	✗	✓
	$<$	$<$	$\geqslant$	✓`execution`	✓	✓
	$\not<$	$<$	$\geqslant$	✓`execution`	✓	✓

Table 1: The class loading behavior exhibited by implementations of aspect-oriented languages. (AspectJ and AspectWerkz exhibit the same behavior and are thus subsumed.)

For all relations of the class loaders l_b, l_c, l_d to an aspect's class loader l_a Table 1 shows whether AspectJ, AspectWerkz, and JAsCo allow or disallow advice weaving. The rightmost column hereby shows the class loading behavior we deem most desirable and which will be argued for in the following. Note that certain combinations are not shown, as the constraints imposed by the Java VM during class loading require that $l_b \preccurlyeq l_c$ and $l_c \succcurlyeq l_d$.

According to the AspectJ language's Development Environment Guide [2] "[a] class loader may only weave classes that it defines." This statement is, however, grossly misleading. As Table 1 shows, execution environments for AspectJ do in many case weave calls when the aspect has been defined by a class loader different from the base program's. The above quote can only be understood in the context of the following rule: "All aspects visible to the weaver are usable. A visible aspect is one defined by the weaving class loader or one of its parent class loaders." Thus, advising a call made by b is allowed whenever $l_a \succcurlyeq l_b$.

As can be seen in the table's first row, each surveyed implementation weaves advice if the aspect's class loader l_a is an ancestor of the other three class loaders l_b, l_c, l_d. However, we deem this behavior undesirable as it allows an aspect a to advise method calls even if symbolic references to methods declared by c are not resolvable since $l_a > l_c$ implies that $l_a \not< l_c$. This behavior then conflicts with the semantics Java as a base language exhibits in the light of reflection: It is not possible for a class a to obtain reflective access to a method declared by c. Weaving is also undesirable in situations where all three class loaders l_b, l_c, l_d are ancestors or siblings of l_a, as this would allow an aspect loaded by l_a to affect the behavior of classes loaded by the bootstrap class loader or comprising an applet container as well as other applets (cf. Figure 1). In fact, none of the surveyed implementations exhibits this behavior.

All other situations shown are desirable and for the most part supported by AspectJ, AspectWerkz, and JAsCo. The situations where $l_a \succcurlyeq l_b$, $l_a < l_c$, and $l_a < l_d$ or $l_a \not< l_d$, however, are only partly supported by AspectJ and AspectWerkz and unsupported by JAsCo. Situations like these occur, e.g., if an applet managed by a container calls to a system class, and said container attempts to advise these calls to introduce access control (cf. Section 2.2). In AspectJ and AspectWerkz this is only possible by means of a **`call`** atomic pointcut, but not by an **`execution`** one, as the former causes advice weaving in the caller's class, whereas the latter weaves into the dynamic callee's class [5]. As JAsCo consistently performs **`execution`**-style weaving [19], this situation cannot be coped with by JAsCo. Similar restrictions apply to situations where $l_a < l_c$, $l_a \geqslant l_d$, and $l_a < l_b$ or $l_a \not< l_b$; they occur, e.g., if an applet registers a call-back with a system class. Hereby, the call-back's interface, i.e., its static type, is defined by the system class loader l_c and the call-back's implementation, i.e., its dynamic type, is defined by the applet's class loader l_d. The situation in the last row is of special interest, as weaving may violate the loading constraints imposed by the Java VM [15, §5.3.4] if the pointcut binds context relating to the (invisible) caller.

The main characteristic of the desired situations shown in Table 1 is that the aspect is visible from either the caller's or the callee's point of view: $l_a \succcurlyeq l_b$ or $l_a \succcurlyeq l_d$; this characterizes not all such situations, however, as the table's first row is exceptional. Furthermore, the table contains only those situations which may occur when adhering to the standard delegation model; pathological cases where, e.g., both $l_a \succ l_b$ and $l_a \prec l_b$ hold have been omitted.[6]

5. Further Issues

As mentioned in Section 3.1, AspectJ also comes with an implementation performing load-time weaving by means of a custom class loader. This implementation has one serious limitation, though: Maintaining multiple namespaces by using a hierarchy of class loaders is impossible. This is due to the fact that advice are woven by the `defineClass` method of the custom `WeavingURLClassLoader`. Thus, in order for a class to be affected, it has to be defined by this loader, which effectively supplants Java's application class loader. However, as the defining class loader, together with the fully qualified name, determines a class's identity, it is outright impossible to use two classes with the same fully qualified name (cf. Section 2.1.2); in a sense, the class loader hierarchy collapses into a single class loader.

[6] The complete data and the survey's setup are available to the public: `http://www.st.informatik.tu-darmstadt.de/static/pages/projects/ALIA/alia.html`.

Another issue is the requirement imposed by the dynamic weavers of both AspectJ and AspectWerkz to declare aspects in an `aop.xml` resource which accompanies the aspect's `class` file. Thus, the question arises whether the class loader l_r loading the resource or the class loader l_a defining the aspect's class determines the protection domain the aspect is assigned to. While it may seem reasonable to enforce the condition $l_a = l_r$, this class loading constraint prevents a valid use case: An aspect in a library defined by an ancestor class loader can selectively be enabled by declaring its use in an `aop.xml` file. Therefore, $l_a \succcurlyeq l_r$ may be an alternative constraint well worth considering.

Finally, one noteworthy difference of JAsCo to the corresponding implementations of AspectJ and AspectWerkz is that JAsCo requires all so-called connectors, which enable JAsCo's aspects, to be loadable by the application class loader. As a direct consequence of this restriction, every aspect resides along with its connectors in the class loader hierarchy at the highest level reachable by ordinary application classes. But this severely limits the possibility to introduce aspect libraries and aspectual containers into the class loader hierarchy.

6. Related Work

While all of the surveyed aspect-oriented languages provide implementations which are aware of the Java VM's class loading facilities, little has been published on the design decisions beyond the level of developer documentation [2]. The need for controlling advice weaving, however, has been acknowledged and led to the proposal of a so-called aspect permission system [8]. The proposed system would ideally extend Java's `AccessController` framework (cf. Section 2.1.1) with permissions controlling whether a particular method call may be advised or not.

While this level of control may be useful, the use of the aforementioned framework requires not only a fine-grained policy, but also restricts the policy to one of "default deny." This generally is a sensible choice. It can be problematic when controlling advice, however, as one of the benefits of AOP is that often the base program can be oblivious of the aspects applied. Such obliviousness would be compromised if permissions needed to be granted to each aspect explicitly. Whether a policy of "default allow" is more desirable is, however, open to debate. In either case, restricting advice weaving by means of the namespace separation offered by Java's dynamic class loading facility nicely complements an aspect permission system as it offers a more coarse-grained mechanism.

7. Conclusion and Future Work

We have shown two shortcomings in the class loading behavior of several existing execution environments for aspect languages based on Java: A protection domain may be erroneously assigned when advice is inlined and namespace separation cannot always be guaranteed. Furthermore, we have identified both desirable and undesirable behavior for dynamic weaving in the presence of class loaders and characterized it analogously to virtual method calls. The desired design has been characterized by two simple conditions based on the static and dynamic type of the callee at join points. In particular, these conditions abstract away from implementation issues like the distinction between **`call`** and **`execution`** join points, which would introduce further subtleties [5] to the already subtle matter of dynamic class loading. We hope that a unified concept, e.g., virtual join points [6], will help to further clarify and formalize the class loading behavior of Java-based aspect-oriented languages. For the moment, however, we propose that the following guidelines be followed when implementing any form of dynamic weaving.

- Advice must be executed within the protection domain of its declaring aspect.
- Advice must resolve symbolic references with the class loader of its declaring aspect.
- Aspects should affect only method calls when visible to the caller's or callee's dynamic type—unless the callee's static type is in turn invisible to the aspect.

However, further design choices still need to be made. We thus seek to establish answers for the following three questions: If aspects are declared and defined by different resources, should constraints on the class loaders be enforced? How to best integrate an aspect permission system with class-based security? And how to secure advice weaving at privileged actions? Answering these questions can guide implementers to aspect languages which fully comply with the security model of the Java language.

Acknowledgments

This work was supported by the AOSD-Europe Network of Excellence, European Union grant no. FP6-2003-IST-2-004349.

References

[1] B. Alpern, C. R. Attanasio, A. Cocchi, D. Lieber, S. Smith, T. Ngo, J. J. Barton, S. F. Hummel, J. C. Sheperd, and M. Mergen. Implementing Jalapeño in Java. In *Proceedings of the 14th Conference on Object-oriented Programming, Systems, Languages, and Applications*, 1999.

[2] The AspectJ Project. *The AspectJ Development Environment Guide*. `http://www.eclipse.org/aspectj/doc/released/devguide/`.

[3] The AspectJ Project. *The AspectJ Programming Guide*. `http://www.eclipse.org/aspectj/doc/released/progguide/`.

[4] P. Avgustinov, A. S. Christensen, L. J. Hendren, S. Kuzins, J. Lhoták, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam,

and J. Tibble. Optimising AspectJ. *ACM SIGPLAN Notices*, 40(6), 2005.

[5] O. Barzilay, Y. A. Feldman, S. Tyszberowicz, and A. Yehudai. Call and execution semantics of AspectJ. In *Proceedings of the 3rd Workshop on Foundations of Aspect-oriented Languages*, 2004.

[6] C. Bockisch, M. Haupt, and M. Mezini. Dynamic virtual join point dispatch. In *Proceedings of the 4th Workshop on Software Engineering Properties of Languages and Aspect Technologies*, 2006.

[7] J. Bonér. AspectWerkz. In *Proceedings of the 3rd Conference on Aspect-oriented Software Development*, 2004.

[8] B. de Win, F. Piessens, and W. Joosen. How secure is AOP and what can we do about it? In *Proceedings of the 2006 Workshop on Software Engineering for Secure Systems*, 2006.

[9] L. Gong, M. Mueller, H. Prafullchandra, and R. Schemers. Going beyond the sandbox: An overview of the new security architecture in the Java development kit 1.2. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, 1997.

[10] J. Gosling, W. N. Joy, G. L. Steele, and G. Bracha. *The Java Language Specification*. Addison-Wesley, 3rd edition, 2005.

[11] E. Hilsdale and J. Hugunin. Advice weaving in AspectJ. In *Proceedings of the 3rd Conference on Aspect-oriented Software Development (AOSD)*, 2004.

[12] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *Proceedings of the 15th European Conference on Object-oriented Programming*, 2001.

[13] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of the 11th European Conference on Object-oriented Programming*, 1997.

[14] S. Liang and G. Bracha. Dynamic class loading in the Java virtual machine. In *Proceedings of the 13th Conference on Object-oriented Programming, Systems, Languages, and Applications*, 1998.

[15] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. The Java Series. Prentice Hall, 2nd edition, 1999.

[16] H. Masuhara and G. Kiczales. Modeling crosscutting in aspect-oriented mechanisms. In *Proceedings of the 17th European Conference on Object-oriented Programming*, 2003.

[17] Sun Microsystems. *The Java HotSpot Server VM*. `http://java.sun.com/products/hotspot/docs/general/hs2.html`.

[18] D. Suvée, W. Vanderperren, and V. Jonckers. JAsCo: an aspect-oriented approach tailored for component based software development. In *Proceedings of the 2nd Conference on Aspect-oriented Software Development*, 2003.

[19] System and Software Engineering Lab, Vrije Universiteit Brussel. *JAsCo language reference 0.8.6*. `http://ssel.vub.ac.be/jasco/lib/exe/fetch.php?media=documentation%3Ajasco.pdf`.

A Decision Tree-based Approach to Dynamic Pointcut Evaluation

Robert Dyer
Department of Computer Science
Iowa State University
rdyer@cs.iastate.edu

Hridesh Rajan
Department of Computer Science
Iowa State University
hridesh@cs.astate.edu

Abstract

Constructs of dynamic nature, e.g., history-based pointcuts and control-flow based pointcuts, have received significant attention in recent aspect-oriented literature. A variety of compelling use cases are presented that motivate the need for efficiently supporting such constructs in language implementations. The key challenge in implementing dynamic constructs is to efficiently support runtime adaptation of the set of intercepted join points at a fine-grained level. This translates to two high-level requirements. First, since the set of intercepted join points may change, such implementations must provide an efficient method to determine this set membership, i.e., whether the currently executing join point needs to be intercepted. Second, the frequency with which such set membership needs to be determined must be minimized. In previous work, Dyer and Rajan proposed a dedicated caching mechanism to address the second requirement. In this work, we propose a mechanism to address the first requirement. This requirement translates to efficiently evaluating whether a join point is intercepted by a set of pointcut expressions. In the worst case, at every join point there may be the need to determine whether it is intercepted. Therefore, even modest savings in such mechanisms is likely to translate to significant savings in the long run.

Categories and Subject Descriptors D.1.5 [*Programming Techniques*]: Object-oriented Programming; D.3.3 [*Programming Languages*]: Language Constructs and Features — Control structures; Procedures, functions, and subroutines; D.3.4 [*Programming Languages*]: Processors — Code generation, Run-time environments

General Terms Algorithms, Design, Languages

Keywords Pointcut evaluation, decision tree, optimization

VMIL'08, October 19, 2008, Nashville, TN, USA.

1. Introduction

In aspect-oriented (AO) languages [9, 15], join points are implicitly-defined by the language as certain kinds of standard actions (such as method calls) in a program's execution. Pointcut designators (PCDs) are used to declaratively select a subset of join points in the program. These selected join points are then composed with additional behavior based on a declarative specification. For this composition (often called weaving), it is necessary to evaluate the PCDs to determine the sub-set of join points that they select. In statically-compiled AO languages, the bulk of the PCD evaluation is done at compile-time and the remaining evaluation is deferred until run-time (often called dynamic residue) [12].

PCD evaluation needs to be deferred until run-time in two cases. First, when the necessary information for PCD evaluation is not known until run-time, e.g., in the case of `if` PCDs where the boolean condition needs to be evaluated at runtime, `this` PCDs where the exact type of the object may not be known statically, `cflow`-like PCDs where the exact control-flow graph may not be known statically, etc. Second, when a new PCD is added to the system, e.g., by dynamically loading a class containing new PCDs and thereby changing the system configuration at runtime [21], by creating new PCDs in more dynamic approaches that support first-class PCDs [30], etc.

A number of techniques have appeared that optimize PCD evaluation for the first case, i.e., when PCD evaluation requires run-time information. Among others, Aotani and Masuhara [2] optimize analysis-based pointcuts, Avgustinov et al. [3] optimize control-flow based pointcuts, Bodden et al. [7] optimize history-based pointcuts, Klose, Ostermann and Leuschel [16] use partial evaluation to reduce the PCD evaluation done at run-time, and most recently Sewe, Bockisch and Mezini [26] optimize evaluation of dynamic residues by eliminating common pointcut expression evaluation. The focus of this paper is optimizing the second case, where new PCDs are added.

Allowing new PCDs to be added to already executing systems is useful for a number of use cases, e.g., in run-time monitoring, run-time adaptation to fix bugs or add features to long running applications, run-time update of dynamic policy changes, etc. AO constructs of dynamic

flavor that fit into these categories are beginning to appear [1, 4–6, 8, 11, 13, 14, 18–21, 23–25, 27–29]. For example, one may want to dynamically modify the behavior of a long-running application (such as a web-server) to start monitoring incoming requests, perhaps after sensing a denial-of-service attack, and then later remove such monitoring once the attack has been thwarted. To model **`cflow`**-like PCDs one may want to start monitoring the likely join points, once the execution reaches the desired entry point in the control-flow, and turn-off monitoring once it reaches the desired exit points [8, 11].

To support such use cases it is important to investigate efficient techniques for runtime PCD evaluation. To that end, this paper makes the following contributions.

- A precise formulation of the PCD evaluation problem and its two different classes that call for different solutions;
- the notion of predicate ordering – based on evaluation cost to optimize PCD evaluation; and
- a decision-tree based technique, corresponding algorithms and data structures for PCD evaluation.

The rest of the paper is organized as follows. In the next section, we formalize the PCD evaluation problem. Section 3 presents our algorithms for PCD evaluation independent of the predicates used in writing PCD expressions. Our method for partially evaluating type predicates is discussed in Section 4. Section 5 discusses related work and Section 6 concludes.

2. PCD Evaluation Problem

In this section, we model the PCD evaluation problem. This is inspired from the formalization of the event matching problem in publish/subscribe systems as described by Fabret et al. [10].

2.1 Terminology

We show the basic terminology used throughout this paper in Figure 1. The definition of *PCD* is fairly straightforward. A *PCD* is either a basic predicate *pred* or a logical conjunction/disjunction of a *pred* and a *PCD*. Note that we do not have negation of a *PCD*, as we assume this is easily emulated using conjunction/disjunction and the operators provided by the various predicates.

A predicate is defined as a 3-tuple (a, o, v): an attribute, an operator and a value. Some example attributes are: modifier(s), return type(s), argument type(s), receiver type, receiver name, method name, control flow, join point kind, etc. The collection of attributes available distinguishes the pointcut expression language. Operators are defined as higher-order functions $o : \mathcal{A} \times \mathcal{V} \rightarrow (\mathcal{A} \times \mathcal{V} \rightarrow \{\texttt{true},\texttt{false}\})$. A predicate can also be thought of as a function (*pred* : $\mathcal{A} \times \mathcal{V} \rightarrow \{\texttt{true},\texttt{false}\}$) obtained by evaluating the expression $o(a, v)$.

$$
\begin{array}{rcl}
\textit{pred} & ::= & (\ a\ ,\ o\ ,\ v\ ) \\
\textit{fact} & ::= & (\ a\ ,\ v\ ) \\
\textit{PCD} & ::= & \textit{pred} \\
 & | & (\ \textit{PCD}\ ) \\
 & | & \textit{pred}\ \&\&\ \textit{PCD} \\
 & | & \textit{pred}\ \|\|\ \textit{PCD} \\
j & ::= & \textit{fact} \\
 & | & \textit{fact}\ \&\&\ j \\
 & & \\
a & \in & \mathcal{A}\text{, the set of attributes} \\
o & \in & \mathcal{O}\text{, the set of operators} \\
v & \in & \mathcal{V}\text{, the set of values}
\end{array}
$$

Figure 1. Basic Terminology

A join point is defined as either a basic *fact* or a logical conjunction of a *fact* and a join point. A *fact* is defined as a pair (a, v) meaning that at the join point the attribute a takes the value v. Evaluating whether a join point satisfies a predicate is equivalent to evaluating each *fact* in the join point w.r.t. the predicate. A *fact* (a', v') satisfies the predicate (a, o, v) if and only if $(o(a, v))(a', v')$ evaluates to **`true`**.

2.2 Example

The following example illustrates the terminology for *PCD* and join points in the context of a small PCD expression language. Let,

- $\mathcal{A} ::= \{modifier, type, name\}$,
- $\mathcal{V} ::= \{v : v$ is a modifier, type or name in the program$\}$, and
- $\mathcal{O} ::= \{==, !=\}$, where the operators have their usual meaning.

An example PCD expression in such a language would be:

```
(modifier, ==, public)                    &&
(type, !=, void) && (name, ==, "Set")
```

and an example join point would be:

```
(modifier, public) && (type, FElement) &&
(name, "Set")
```

2.3 PCD Evaluation

For a program, let $\mathcal{J}$ be the set of join points (possibly unknown statically) and $\mathcal{P}$ be the set of *PCD* expressions as defined previously. We present two alternative formulations of the PCD evaluation problem ($PCDEval$).

1. Given a join point and a set of pointcut expressions, we are interested in determining the subset of pointcut expressions that may select this join point. Formally, $PCDEval : \mathcal{J} \times 2^{\mathcal{P}} \rightarrow 2^{\mathcal{P}}$, where $2^{\mathcal{P}}$ is the power set of $\mathcal{P}$.
2. Given a pointcut and a set of join points, we are interested in determining the subset of join points that are selected

by such a pointcut. Formally, $PCDEval' : \mathcal{P} \times 2^{\mathcal{J}} \rightarrow 2^{\mathcal{J}}$, where $2^{\mathcal{J}}$ is the power set of $\mathcal{J}$.

These formulations are useful for compile-time, load-time, and runtime pointcut evaluation. For example, most AO compilers today use the first formulation for weaving. The rationale is that often the total number of join points is larger compared to the total number of pointcuts and an efficient solution to the first formulation may reduce the number of iterations through the set of join points. This formulation also fits better with aspect-oriented systems that allow load-time deployment with a closed-world assumption for aspects, such as the AspectJ's load-time weaver. Here, by closed-world assumption for the set of aspects we mean that all aspects are loaded before any class is loaded. During class loading, each join point shadow in the class is matched with the set of pointcut expressions. Furthermore, aspect-oriented systems that allow run-time deployment, such as the *Nu* virtual machine [8], can also utilize this strategy. In *Nu* for example, a lazy strategy is used for run-time weaving, where a join point is not matched until it executes at least once.

On the other hand, for incremental compilation of AO programs, in cases where the increments introduce new *PCD* expressions, it would perhaps be more appropriate to use efficient solutions to the second formulation ($PCDEval'$). Load-time weavers with an open world assumption for aspects would also benefit from the second formulation, if they employ an eager strategy for matching. Such an eager strategy would match the PCDs just loaded with join point shadows in all classes already loaded, perhaps to avoid matching overhead during execution. Similarly, runtime weaving systems can also utilize $PCDEval'$ for cases where deployed aspects affect the "hot" segments and they are unlikely to be un-deployed. Other formulations can also be conceived that perhaps take a hybrid approach, but for the purpose of this paper we will not consider them.

3. PCD Evaluation Algorithm

This section describes our approach for PCD evaluation. The minimum requirement is fairly straightforward: the worst-case time complexity of our technique should not exceed the time complexity of current PCD evaluation methods. An additional requirement that we impose on our technique is that its amortized complexity should be independent of the number of PCDs in the system. Note that in this paper we are only considering the first formulation $PCDEval : \mathcal{J} \times 2^{\mathcal{P}} \rightarrow 2^{\mathcal{P}}$.

3.1 Predicate Ordering

The first step in our PCD evaluation technique is to order the evaluation of predicates in the PCD. Note that a PCD consists of conjunction and/or disjunction of one or more predicates. To determine whether a join point is selected by a PCD, it is necessary to determine whether there exists a satisfiable assignment of the predicate(s) at the join point for which the PCD evaluates to true.

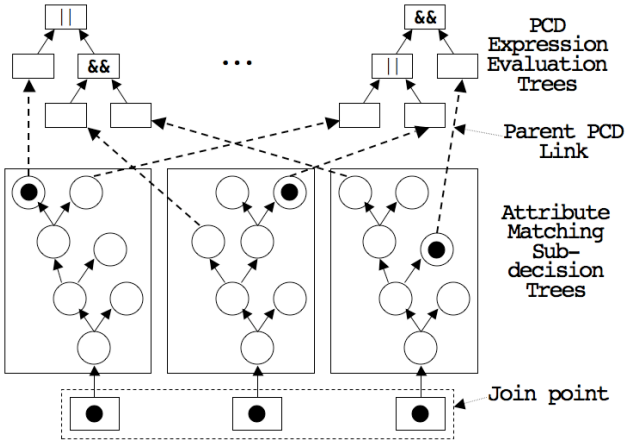


Figure 2. An Overview of our Decision Tree-based Approach for PCD Evaluation

Let $\mathcal{C}$ be an amortized cost function such that $\mathcal{C}(a_i)$ is the amortized cost of evaluating the attribute a_i over operators defined for a and the set of values for a_i. The first step in our technique is to order the evaluation of each $a_i, a_j \in \mathcal{A}$ such that a_i is evaluated prior to a_j, if $\mathcal{C}(a_i) < \mathcal{C}(a_j)$. If $\mathcal{C}(a_i) = \mathcal{C}(a_j)$ ordering may be determined heuristically.

The rationale for adopting this policy for predicate evaluation ordering is to decrease the amortized cost of PCD evaluation by eliminating as many PCDs as possible at a lower cost. This strategy goes back to the efficient ordering of boolean predicate evaluation in SAT solvers. It has also recently been applied by Sewe, Bockisch, and Mezini [26] for optimizing evaluation of dynamic residues.

3.2 Data Structures for PCD Evaluation

An overview of the data structures maintained for our PCD evaluation algorithm is shown in Figure 2. From the top to bottom, the figure shows the set of PCD expression evaluation trees, sub-trees for matching attributes in the pointcut expression language, and a join point (a logical conjunction of facts) being matched. The number of sub-decision trees depend on the types of attributes available in the pointcut expression language. For example, a pointcut expression language that only allows matching based on types would just have one such decision tree.

For languages that provide different kinds of join points, e.g. **`execution`**, **`handler`** and **`set`**, in an AspectJ-like language, it would be sensible to maintain this data structure separately for each join point kind, as these would be disjoint. Furthermore, for each join point kind it would be appropriate to customize the set of attribute sub-decision trees, e.g., decision trees for name and type for **`set`** and **`get`** join points.

The PCD expressions are organized into a forest of PCD expression evaluation trees. These trees may have cross-links. These cross-links are created for common-subexpression elimination. Addition of a new PCD to this forest proceeds as follows:

(a) add the component predicates of the PCD to their respective attribute decision tree,
(b) add the tree representation of the PCD to the forest, and
(c) create a parent PCD link between attribute nodes and the leaf node of the PCD expression evaluation tree (shown as bold dotted arrows in the figure).

Removing a PCD is the reverse of this process, except that optimizations due to common subexpression elimination must be taken into account. For this purpose, a simple reference count is maintained that reflects the number of PCDs that contain this attribute node or a PCD expression sub-tree as a parent.

3.3 Optimizations of PCD Expression Tree

We assume that the PCD expressions are locally optimized before being added to the PCD expression evaluation forest. For example, parts of a PCD that will never match are eliminated, common local subexpressions are eliminated, etc. Sewe, Bockisch, and Mezini [26] discuss some of these techniques.

We also optimize PCD expressions by reorganizing the PCD expression trees. An example reorganization is shown in Figure 3, where the OR operator is successively propagated downward.

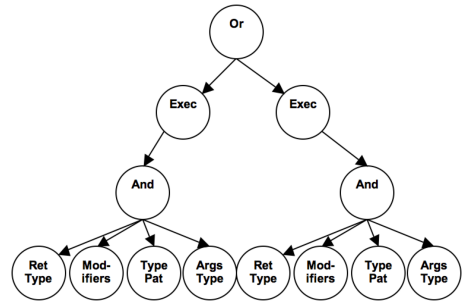
The reorganization is done using post-order tree-traversal technique and it terminates when a *classifier attribute* is the root node of every PCD expression tree. *Classifiers* are attributes that help pigeonhole PCD expression trees into a disjoint subset of join points. For example, the join point shadow kind is a type of classifier as it helps categorize the PCD expression tree into different classes based on which join point kind they match.

The reorganization of PCD expression trees has three benefits:

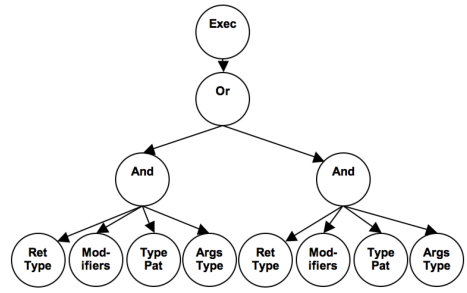
1. It helps reduce the depth of the PCD expression tree,
2. it helps classify PCD expression trees into disjoint sets for which separate PCD expression forests could be maintained, and
3. it enables elimination of certain PCD expressions by partial evaluation.

We will discuss more partial evaluation strategies in Section 4. These three benefits directly translate to decrease in the PCD evaluation overhead potentially reducing the runtime overhead of dynamic deployment of aspects.

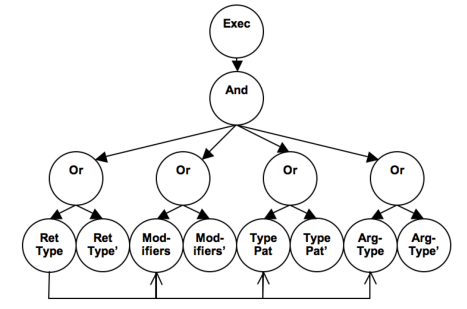
There are no general techniques for maintaining the decision tree for each attribute. Instead it depends very much on the kind of the attribute. Efficient matching of modifiers, for example, requires completely different data structures and algorithms compared to matching of names and types. In this paper, we discuss algorithms and data structures for some of these, but we do not attempt to be exhaustive.



(a) PCD expression tree for execution(..)||execution(..)



(b) OR operator propagated downward in the tree



(c) OR operator propagated further down

Figure 3. Reorganizing pattern tree by propagating OR operator downward

3.4 Matching of Join Point Facts

A join point (or the conjunction of facts about a join point) is matched against this combination of PCD expression forest and attribute decision trees. The PCD evaluation starts with lowest cost attribute as discussed in Section 3.1. On traversing the attribute decision tree, at each node decisions are made about whether the current fact about the join point implies the predicate represented by that node.

If the predicate represented by the current node is implied, a *token* is sent to the leaf nodes of each parent PCD expression that contains that predicate as a component. The

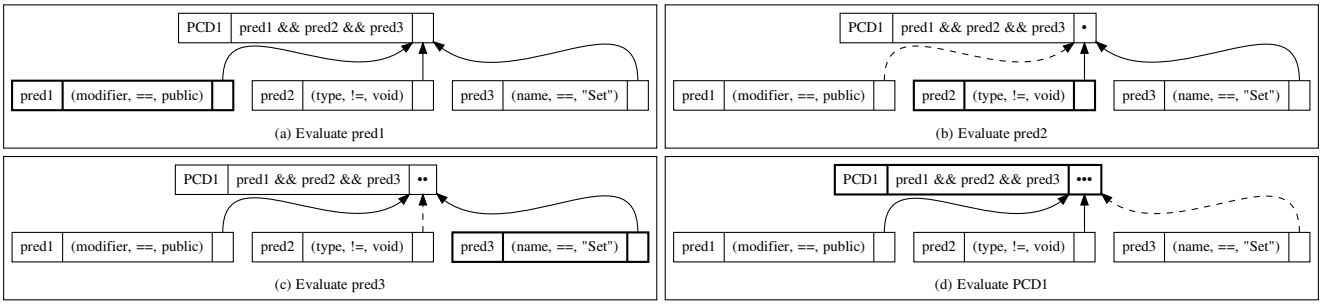


Figure 4. Example of matching the join point $(modifier, public)$ && $(type, FElement)$ && $(name, \text{“}Set\text{”})$. Dotted lines represent tokens being sent to a parent node. Bold boxes indicate the predicate being evaluated at that step.

leaf nodes in the PCD expression send tokens up the PCD expression forest depending on whether their parent node is a conjunction or a disjunction node.

While traversing the attribute decision tree, the algorithm keeps track of whether any tokens have been sent to the PCD expression forest. When the traversal of the attribute decision tree is complete, i.e., a leaf node in the decision tree is reached, if no tokens are sent this far to the PCD expression forest, the PCD evaluation terminates. This helps ensure that the least costly attributes often help short-circuit PCD evaluation.

A simple example of PCD evaluation is given in Figure 4. In this example, there is one *PCD* in the system. The *PCD* is $(modifier, ==, public)$ && $(type, !=, void)$ && $(name, ==, \text{“}Set\text{”})$. The join point we are trying to match is $(modifier, public)$ && $(type, FElement)$ && $(name, \text{“}Set\text{”})$.

At each step of the example we are evaluating one *pred* or *PCD* (indicated with a bold box). Consider step (a), where we are evaluating the *pred* $(modifier, ==, public)$. Since the join point contains $(modifier, public)$ this evaluates to **`true`**. The *pred* then sends a token to each parent node (in this case, there is one parent node – PCD1). The action of a token moving to another node is shown with a dashed line. In steps (b) and (c), pred2 and pred3 evaluate to **`true`** and similarly each sends a token.

The final step is (d), where PCD1 is evaluated. Since PCD1 is a conjunction of three *pred*'s, in order to evaluate to **`true`** it must contain three tokens. In this example it does, so PCD1 would evaluate to **`true`** meaning that the join point being matched matches PCD1.

4. Partial Evaluation of Type Predicates

In this section, we describe our techniques for partial evaluation of type predicates. The key idea behind our partial evaluation technique is to utilize the implication relationships between types. These partially evaluated predicates are then organized as a decision tree that helps optimize runtime evaluation. In the rest of this section, we describe various aspects of our technique. First, the partial evaluation results for logical operators are defined. We then describe a simple data structure and efficient algorithms that utilize these partial evaluation results.

4.1 Semantics of Type Operators

Let us suppose $A, B, C, \ldots$ be the types in the program and *op* be the type operator, where $op \in \{\lessdot, \gtrdot, \doteq, \neq\}$ such that:

- $\boldsymbol{A \lessdot B}$ means that A is a strict subtype of B, i.e., it excludes the case when $A \doteq B$ (see below).
- $\boldsymbol{A \gtrdot B}$ means that A is a strict super-type of B, i.e., it excludes the case when $A \doteq B$ (see below).
- $\boldsymbol{A \doteq B}$ means that A is exactly of the same type as B.
- $\boldsymbol{A \neq B}$ means that A is not of the same type as B. In addition, A is not a strict subtype of B, and B is not a strict subtype of A.

Note that the meaning of the type operators is slightly different from the standard definitions. The operators are defined in this manner to facilitate partitioning the type predicates into disjoint subsets. We have used slightly different symbols to remind readers of the difference.

4.2 Semantics of logical inverse on type operators

Our technique for partially evaluating type predicates relies on computing the inverse of a predicate. For simplicity we compute the inverse of a predicate by inverting the operator. For example, the inverse of $(A \lessdot B)$ is computed by inverting the $\lessdot$ operation. Below we define the inverse of type operators. Let us assume that ! is the inverse operator such that $!(A\ op\ B)$ is defined as:

- $!(A \lessdot B) \equiv A \gtrdot B \vee A \doteq B \vee A \neq B$
- $!(A \gtrdot B) \equiv A \lessdot B \vee A \doteq B \vee A \neq B$
- $!(A \doteq B) \equiv A \lessdot B \vee A \gtrdot B \vee A \neq B$
- $!(A \neq B) \equiv A \lessdot B \vee A \gtrdot B \vee A \doteq B$

4.3 Partial evaluation of logical conjunctions (and) on type predicates

Let $\wedge$ be the logical conjunction operator, let $op = op'$ be the equality operator where $op, op' \in \{\lessdot, \gtrdot, \doteq, \neq\}$, and let T and F be the Boolean truth values with standard meanings.

Let $\mathcal{J}_1$ = (type, A) be a **fact** in the system where A is a type. Let $\mathcal{P}_1$ = (type, op_1, B) and $\mathcal{P}_2$ = (type, op_2, C) be type predicates where B and C are types. To see if the **fact** matches both predicates, we thus want to evaluate $(A\ op_1\ B) \wedge (A\ op_2\ C)$.

In order to partially evaluate this expression, we use a given fact $B\ op\ C$ and derive implication rules among the types B and C. Figure 5 shows the results for all four operators.

As an example of how we derived these rules, consider the case $B \lessdot C$ when $op_1 == \lessdot$ and $op_2 == \lessdot$. Thus we are interested in evaluating $A \lessdot B \wedge A \lessdot C$. If we assume $A \lessdot B$, since we are given $B \lessdot C$ we can see this implies $A \lessdot C$.

Now consider the case $B \lessdot C$ when $op_1 == \lessdot$ and $op_2 == \gtrdot$. Thus we are interested in evaluating $A \lessdot B \wedge A \gtrdot C$. If we assume $A \lessdot B$, since we are given $B \lessdot C$ we already showed this implies $A \lessdot C$. This would contradict $A \gtrdot C$ and thus this reduces to **`false`**.

Most cases are easily derived in a similar fashion, and thus omitted for space. We will discuss one interesting case. Again we have $B \lessdot C$. When $op_1 = \gtrdot$ and $op_2 = \neq$ we are interested in evaluating $A \gtrdot B \wedge A \neq C$. This expression can not be reduced. $A \gtrdot B \wedge B \lessdot C$ implies that B is a subtype of both A and C. $A \neq C$ implies that neither A is a subtype of C, nor C is a subtype of A. Therefore, in the semantics of single inheritance languages this would evaluate to F; however, in the semantics of languages that support limited multiple inheritance such as through interfaces in Java this logical conjunction may not be reduced.

The rules for the $\gtrdot$ operator is the mirror image of the $\lessdot$ operator along the main diagonal line. The rules for the $\doteq$ operator are easily derived, since anything not on the diagonal is clearly a contradiction and evaluates to F. For everything else, we can simply choose either of the two **facts**, as they are actually identical. Derivation of the rules for the $\neq$ operator are omitted for space reasons.

4.4 Attribute Decision Tree for Types

In this section, we discuss the algorithms and data structures for maintaining a decision tree for types. A pointcut expression language can employ such decision tree for matching return type, receiver type and argument types of methods, constructors, etc as part of join points of kind **`execution`**, **`call`**, **`initialization`**, etc, for types of fields for join points of kind **`set`**, **`get`**, for types of exception for join points of kind **`handler`**, just to name a few. Thus, the data structures and algorithms for this attribute are likely to be helpful in the implementation of PCD evaluation for common PCD expression languages.

This attribute decision tree could also be useful for VM-based implementations of languages that match purely based on types, such as Ptolemy [22].

We first discuss a technique for adding a type predicate to the decision tree. This technique makes use of a partial-evaluation function to optimize the matching process. This partial evaluation function was described in the previous section. We then discuss a technique for matching type-related facts about the join point using this decision tree. As previously discussed, it would be sensible to maintain separate copies of this decision tree for each kind, e.g., return type, argument type, and receiver type.

4.4.1 Addition of Predicates to Type Decision Tree

Our algorithm for adding a type predicate to an existing decision tree is shown in Algorithm 1 and explained below.

Algorithm 1: Insert: Adds a Type Predicate to the Predicate Tree

```
   Input: Predicate tree: Tree, Predicate: Pred
 1 Current = Tree.Root;
 2 if Pred == true then
 3     Current.Parents.Append(Pred);
 4     return
 5 while Current.TrueBranch != NULL do
 6     Current = Current.TrueBranch;
 7     if Current == Pred then
 8         Current.Parents.Append(Pred);
 9         return
10     else
11         result = PartialEval (Current ∧ Pred);
12         if result == Current then
13             return
14         if result == Pred then
15             Swap (Current,Pred);
16             return
17         if result == false then
18             if Current.FalseBranch == NULL then
19                 Current.FalseBranch = new
                   Node(Pred);
20                 return
21             else
22                 Current = Current.FalseBranch
23 end
24 Current.TrueBranch.Parents.Append(Pred);
```

The addition of a type predicate to the predicate tree is an incremental process that starts with the root node of the current tree and the predicate that is to be added. The type predicate evaluation tree is maintained as a binary tree with two branches labeled `TrueBranch` and `FalseBranch`

$\wedge$	$op_2 = \lessdot$	$op_2 = \gtrdot$	$op_2 = \doteq$	$op_2 = \neq$
$op_1 = \lessdot$	$A \lessdot B$	F	F	F
$op_1 = \gtrdot$	$(A \gtrdot B) \wedge (A \lessdot C)$	$A \gtrdot C$	$A \doteq C$	F or $(A \gtrdot B) \wedge (A \neq C)$
$op_1 = \doteq$	$A \doteq B$	F	F	F
$op_1 = \neq$	$(A \neq B) \wedge (A \lessdot C)$	F	F	$(A \neq B) \wedge (A \neq C)$

(a) Case: $B \lessdot C$

$\wedge$	$op_2 = \lessdot$	$op_2 = \gtrdot$	$op_2 = \doteq$	$op_2 = \neq$
$op_1 = \lessdot$	$A \lessdot C$	$(A \gtrdot C) \wedge (A \lessdot B)$	$A \doteq C$	$(A \neq C) \wedge (A \lessdot B)$
$op_1 = \gtrdot$	F	$A \gtrdot B$	F	F
$op_1 = \doteq$	F	$A \doteq B$	F	F
$op_1 = \neq$	F	F or $(A \gtrdot C) \wedge (A \neq B)$	F	$(A \neq C) \wedge (A \neq B)$

(b) Case: $B \gtrdot C$ - As expected, the partial evaluation matrix in this case is the mirror image of the matrix for $B \lessdot C$ along the main diagonal.

$\wedge$	$op_2 = \lessdot$	$op_2 = \gtrdot$	$op_2 = \doteq$	$op_2 = \neq$
$op_1 = \lessdot$	$A \lessdot B$	F	F	F
$op_1 = \gtrdot$	F	$A \gtrdot B$	F	F
$op_1 = \doteq$	F	F	$A \doteq B$	F
$op_1 = \neq$	F	F	F	$A \neq B$

(c) Case: B $\doteq$ C

$\wedge$	$op_2 = \lessdot$	$op_2 = \gtrdot$	$op_2 = \doteq$	$op_2 = \neq$
$op_1 = \lessdot$	$(A \lessdot B) \wedge (A \lessdot C)$	F	F	$(A \lessdot B) \wedge (A \neq C)$
$op_1 = \gtrdot$	F	$(A \gtrdot B) \wedge (A \gtrdot C)$	F	$(A \gtrdot B) \wedge (A \neq C)$
$op_1 = \doteq$	F	F	F	$A \doteq B$
$op_1 = \neq$	$(A \neq B) \wedge (A \lessdot C)$	$(A \neq B) \wedge (A \gtrdot C)$	$A \doteq C$	$(A \neq B) \wedge (A \neq C)$

(d) Case: B $\neq$ C

Figure 5. Partial Evaluation Rules for Type Predicates

(except for the root node as described below). Both these branches need to be present at all time.

The current tree is traversed until an appropriate position for the current predicate is found. The root node of the tree represents the predicate **`true`** and it trivially matches any fact during the matching. As a special case all value types go to the false branch and all reference types go to the true branch of root.

All predicates `ret <: object` are trivially implied, if we are traversing the reference type branch. Therefore, if the predicate being added is that, it is simply appended to the root of the reference subtree. In particular, a parent PCD link is created from this node to the leaf node of PCD expression tree such that whenever this predicate evaluates to true parent PCDs can be notified. This step facilitates common-predicate elimination.

The algorithm terminates when the true branch of the root node does not exist. The new predicate is then added to the true branch of the root node.

If the true branch exists and the added predicate is not the trivially implied predicate `ret <: object` the main loop of the algorithm begins that continues until the current predicate being explored evaluates to **`null`**.

In this loop, implication relationships are used to determine the branch of the decision tree traversed. These relationships are computed using our partial evaluation rules for types as shown in Section 4. The function `PartialEval` facilitates that. If the result of this function is the current predicate `Current` or the predicate being added `Pred`, it means that true/false evaluation of one predicate implies true/false evaluation of the other. This works in general because implication is a transitive relation.

4.4.2 Simultaneous Evaluation of Predicates in the Type Decision Tree

Our algorithm for matching a type-related fact in an existing decision tree is shown in Algorithm 2 and explained below.

The evaluation of a type predicate tree against a fact starts at the root node of the tree. Note that the root node represents the predicate **`true`** and therefore any fact trivially matches this predicate. If there are complex predicates that contain true type predicates they are immediately notified.

If the fact implies the current predicate being evaluated, the current predicate and all predicates implied by it are automatically evaluated to be true and the matching algorithm terminates. If on the other hand the fact does not imply the current predicate being evaluated, we compute the relation-

Algorithm 2: Match - Evaluate a fact against a predicate tree, resulting in tokens at predicates that evaluate true for the fact

```
   Input: Predicate tree: Tree, Fact: fact
 1 Current = Tree.Root;
 2 NotifyParents (Current);
 3 while Current != NULL do
 4     result = PartialEval (Current ∧ fact);
 5     if result == Current then
 6         NotifyParents (Current);
 7         return
 8     if result == Pred then
 9         Current = Current.TrueBranch
10     else
11         Current = Current.FalseBranch
12     if fact == Current then
13         NotifyParents (Current);
14         return
15 end
```

ships between the type value in the fact `fact.Value` and the type value in current predicate `current.Value` to minimize matching. In particular, we evaluate the logical relationships that exist between these types (e.g., strict subtype of, strict supertype of, etc) as defined in Section 4.1.

The logical relationship between the type value in the fact and the type value in the current predicate is then used to lookup the partial evaluation results. We will describe these in details in later section. For understanding this algorithm, it is sufficient to know that the looking up statically computed partial evaluation results returns three different results. First, that suggests that the fact implies current predicate. Second, that suggests that the current predicate implies the fact. Third, that suggests that the fact may never imply the current predicate or vice-versa and fourth, that suggests that these values cannot be partially reduced.

The first case implies that the current predicate and all predicates implied by it will evaluate to **`true`** for the fact being matched. The second case implies that even though the current predicate will not evaluate to **`true`** for this fact, only predicates in its true subtree may evaluate to **`true`** (by construction), therefore only exploring the true subtree of the current predicate will be sufficient. The third and the fourth case imply that the current predicate and all predicates in its true branch will not evaluate to **`true`** for this fact, therefore only exploring the false subtree of the current predicate will be sufficient.

4.4.3 Implementations for Java

Very fast mechanisms exist for computing logical relationships between types such as the implementation of the **`instanceof`** construct in Java. The relationship information that we require can be computed with an **`instanceof`** and an `equals` comparison. Further discussion is beyond the scope of this paper, but it suffices to say that for a further reduced cost, an operator can also be implemented in the virtual machine that utilizes the information maintained for efficiently computing the **`instanceof`** relationships to compute these logical relationships at the cost of an **`instanceof`** operator.

There are two optimizations (not shown in Algorithm 2) implemented for languages such as Java, that support primitive value types and reference types and a top type `object`. The type predicate tree maintains a subtree for primitive value types and another subtree for reference types. If the fact is a value type, the evaluation proceeds with the value type subtree otherwise the reference type subtree is explored. Furthermore, all facts that proceed to match the reference type subtree, implicitly match the top type `object`.

5. Related Work

Recently, Sewe et al. described a method of using ordered binary decision diagrams (BDD) to eliminate redundant evaluations of dynamic residues [26]. Dynamic residues are the result of compilers statically performing partial evaluation on the pointcuts [17]. By converting the residues into an ordered BDD, they are able to evaluate the dynamic residues of all pointcuts for a given join point while only evaluating each atomic residue at most once.

Similar to our technique, they also order the evaluation of the atomic pointcuts using the cost of their evaluations for improved efficiency. Our matching technique however does not focus on the dynamic residues left over from previous partial evaluation of pointcuts by compilers. Instead, it focuses on dynamically evaluating the set of full pointcuts in the system against a join point. Both approaches do however make use of decision trees during the matching process.

Previous work by Klose et al. has shown that the use of partial evaluation techniques can reduce the amount of work needed to match a PCD at runtime [16]. Their specializer generates efficient checks for the program, however this is done offline. Similar to their approach, we use partial evaluation techniques to try and minimize the cost of matching a PCD, however our partial evaluation is performed online using dynamic information about the classes in the system. Thus, while we incur an overhead of performing the partial evaluation at runtime, we potentially have more information available for even more efficient PCD matching. This trade-off is most beneficial in systems where the set of PCDs changes often or systems that execute for a long period of time.

6. Conclusion and Future Work

The need for efficient support of dynamic aspect-oriented constructs dictates that more efficient techniques are provided in virtual machines for PCD evaluation. The use cases

for dynamic aspect-oriented constructs are attractive and with the availability of more efficient implementations, more applications for such constructs can be explored, where concerns about overhead are an important factor in adoption. The decision-tree based technique for PCD evaluation that we present in this work seems promising in that regard, although a rigorous evaluation is needed to exactly characterize the benefits in terms of space and time complexity. In particular, it will be interesting to study the following parameters.

1. Total number of predicates in the system: Studying this parameter will show how the performance scales with the total number of predicates in the system.
2. Sub-classing (sub-typing) relationship: how many predicate values in the system are in the type hierarchy of other predicate values, i.e., strict super type of, strict subtype of, or equal.
3. Unrelated types: how many predicate values in the system are not related to other predicate values?
4. True predicates: studying this parameter will show how pointcuts that use a significant amount of wild-cards influence the performance of the type decision tree.
5. Percentage of successful matches: this parameter will show how successful matches contribute to the cost of performance evaluation. It will also help characterize the one time cost paid by the join points that match in a dynamic AO language model.
6. Percentage of unsuccessful matches: study of this parameter will show how unsuccessful matches contribute to the cost of performance evaluation. This will help determine the one time cost paid by the join points that do not match in a dynamic AO language model. In particular, the sooner the decision tree can determine that the join point is not going to match, the better.

Acknowledgments

This work is supported in part by Iowa State University's generous startup grant and the NSF grants CNS-06-27354 and CNS-07-09217. Thanks to the anonymous reviewers of the VMIL workshop for their comments.

References

[1] Chris Allan, Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Ondřej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. Adding trace matching with free variables to AspectJ. In *OOPSLA '05: Proceedings of the 20th international conference on Object-Oriented Programs, Systems, Languages, and Applications*, New York, NY, USA, 2005. ACM Press.

[2] Tomoyuki Aotani and Hidehiko Masuhara. SCoPE: an AspectJ compiler for supporting user-defined analysis-based pointcuts. In *AOSD '07: Proceedings of the 6th international conference on Aspect-Oriented Software Development*, pages 161–172, New York, NY, USA, 2007. ACM Press.

[3] Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Jennifer Lhoták, Ondřej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. Optimising AspectJ. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming Language Design and Implementation*, pages 117–128, New York, NY, USA, 2005. ACM Press.

[4] Jason Baker and Wilson Hsieh. Runtime aspect weaving through metaprogramming. In *AOSD '02: Proceedings of the 1st international conference on Aspect-Oriented Software Development*, pages 86–95, New York, NY, USA, 2002. ACM Press.

[5] Christoph Bockisch, Matthew Arnold, Tom Dinkelaker, and Mira Mezini. Adapting virtual machine techniques for seamless aspect support. In *OOPSLA '06: Proceedings of the 21st international conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 109–124, New York, NY, USA, 2006. ACM Press.

[6] Christoph Bockisch, Michael Haupt, Mira Mezini, and Klaus Ostermann. Virtual machine support for dynamic join points. In *AOSD '04: Proceedings of the 3rd international conference on Aspect-Oriented Software Development*, pages 83–92, New York, NY, USA, 2004. ACM Press.

[7] Eric Bodden, Laurie Hendren, and Ondřej Lhoták. A staged static program analysis to improve the performance of runtime monitoring. In *ECOOP '07: Proceedings of the 21st European Conference on Object-Oriented Programming*, pages 525–549. Springer-Verlag, 2007.

[8] Robert Dyer and Hridesh Rajan. Nu: a dynamic aspect-oriented intermediate language model and virtual machine for flexible runtime adaptation. In *AOSD '08: Proceedings of the 7th international conference on Aspect-Oriented Software Development*, New York, NY, USA, 2008. ACM Press.

[9] Tzilla Elrad, Robert E. Filman, and Atef Bader. Aspect-oriented programming: Introduction. *Communications of the ACM*, 44(10):29–32, 2001.

[10] Françoise Fabret, H. Arno Jacobsen, François Llirbat, João Pereira, Kenneth A. Ross, and Dennis Shasha. Filtering algorithms and implementation for very fast publish/subscribe systems. In *SIGMOD '01: Proceedings of the 2001 international conference on Management of Data*, pages 115–126, New York, NY, USA, 2001. ACM.

[11] Stefan Hanenberg, Robert Hirschfeld, and Rainer Unland. Morphing aspects: incompletely woven aspects and continuous weaving. In *AOSD '04: Proceedings of the 3rd international conference on Aspect-Oriented Software Development*, pages 46–55, New York, NY, USA, 2004.

[12] Erik Hilsdale and Jim Hugunin. Advice weaving in AspectJ. In *AOSD '04: Proceedings of the 3rd international conference on Aspect-Oriented Software Development*, pages 26–35, New York, NY, USA, 2004. ACM Press.

[13] Robert Hirschfeld. AspectS - aspect-oriented programming with Squeak. In *NODe '02: Revised Papers from the international conference NetObjectDays on Objects, Components, Architectures, Services, and Applications for a Networked World*, pages 216–232, London, UK, 2003. Springer-Verlag.

[14] Robert Hirschfeld and Stefan Hanenberg. Open aspects.

Computer Languages, Systems & Structures, 32(2-3):87–108, 2006.

[15] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *ECOOP '97: Proceedings of the 11th European Conference on Object-Oriented Programming*, Finland, June 1997. Springer-Verlag.

[16] Karl Klose, Klaus Ostermann, and Michael Leuschel. Partial Evaluation of Pointcuts. In *PADL '07: Proceedings of the 9th international symposium on Practical Aspects of Declarative Languages*, volume 4354, pages 320–334. Springer-Verlag, 2007.

[17] Hidehiko Masuhara, Gregor Kiczales, and Chris Dutchyn. A compilation and optimization model for aspect-oriented programs. In *CC '03: Proceedings of the 12th conference on Compiler Construction*, pages 46–60. Springer-Verlag, 2003.

[18] Francisco Ortin and Juan Manuel Cueva. Dynamic adaptation of application aspects. *Journal of Systems and Software*, 71(3):229–243, 2004.

[19] Renaud Pawlak, Lionel Seinturier, Laurence Duchien, and Gerard Florin. JAC: A flexible solution for aspect-oriented programming in Java. In *REFLECTION '01: Proceedings of the 3rd international conference on Metalevel Architectures and Separation of Crosscutting Concerns*, pages 1–24, London, UK, 2001. Springer-Verlag.

[20] Andrei Popovici, Gustavo Alonso, and Thomas Gross. Just-in-time aspects: efficient dynamic weaving for Java. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-Oriented Software Development*, New York, NY, USA, 2003. ACM Press.

[21] Andrei Popovici, Thomas Gross, and Gustavo Alonso. Dynamic weaving for aspect-oriented programming. In *AOSD '02: Proceedings of the 1st international conference on Aspect-Oriented Software Development*, pages 141–147, New York, NY, USA, 2002. ACM Press.

[22] Hridesh Rajan and Gary T. Leavens. Ptolemy: A language with quantified, typed events. In *ECOOP '08: Proceedings of the 22nd European Conference on Object-Oriented Programming*. Springer-Verlag, July 2008.

[23] Hridesh Rajan and Kevin J. Sullivan. Eos: instance-level aspects for integrated system design. In *ESEC/FSE-11: Proceedings of the 9th European Software Engineering Conference held jointly with 11th international symposium on Foundations of Software Engineering*, pages 297–306, New York, NY, USA, 2003. ACM Press.

[24] Hridesh Rajan and Kevin J. Sullivan. Need for instance level aspect language with rich pointcut language. In *SPLAT '03: Software engineering Properties of Languages for Aspect Technologies*, 2003.

[25] Hridesh Rajan and Kevin J. Sullivan. Classpects: unifying aspect- and object-oriented language design. In *ICSE '05: Proceedings of the 27th International Conference on Software Engineering*, pages 59–68, New York, NY, USA, 2005. ACM Press.

[26] Andreas Sewe, Christoph Bockisch, and Mira Mezini. Redundancy-free residual dispatch. In *FOAL '08: Foundations of Aspect-Oriented Languages workshop*, 2008.

[27] Volker Stolz and Eric Bodden. Temporal assertions using AspectJ. In *RV '05: 5th workshop on Runtime Verification*, 2005.

[28] Volker Stolz and Eric Bodden. Tracechecks: Defining semantic interfaces with temporal logic. *Software Composition*, pages 147–162, 2006.

[29] Davy Suvée, Wim Vanderperren, and Viviane Jonckers. JAsCo: an aspect-oriented approach tailored for component based software development. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-Oriented Software Development*, pages 21–29, New York, NY, USA, 2003. ACM Press.

[30] David B. Tucker and Shriram Krishnamurthi. Pointcuts and advice in higher-order languages. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 158–167. ACM, 2003.