FOAL 2002 Proceedings
Foundations of Aspect-Oriented Languages
Workshop at AOSD 2002

Gary T. Leavens and Ron Cytron (editors)
TR #02-06
April 2002

Keywords: Aspect-oriented programming languages, formal semantics, formal methods, specification, verification, theory of testing, theory of aspect composition, aspect translation and rewriting, compilation, advice, join points, member-group relations, superposition, observers, assistants, modularity, events, source-code instrumentation.

2000 CR Categories: D.1.m [*Programming Techniques*] Miscellaneous — aspect-oriented programming, reflection; D.2.1 [*Software Engineering*] Requirements/Specifications — languages, methodology, theory, tools; D.2.4 [*Software Engineering*] Software/Program Verification — class invariants, correctness proofs, formal methods, programming by contract, reliability, validation; D.3.1 [*Programming Languages*] Formal Definitions and Theory — semantics; D.3.3 [*Programming Languages*] Language Constructs and Features — control, data types and structures; F.3.1 [*Logics and Meaning of Programs*] Specifying and verifying and reasoning about programs — assertions, logics of programs, pre- and post-conditions, specification techniques; F.3.m [*Logics and Meaning of Programs*] Miscellaneous — reasoning about performance.

Department of Computer Science
226 Atanasoff Hall
Iowa State University
Ames, Iowa 50011-1040, USA

Table of Contents

Preface

Aspect-oriented programming is a new area in software engineering and programming languages that promises better support for separation of concerns. The first Foundations of Aspect-Oriented Languages (FOAL) workshop was held at the 1st International Conference on Aspect-Oriented Software Development in Enschede, The Netherlands, on April 22, 2002. This workshop was designed to be a forum for research in formal foundations of aspect-oriented programming languages. The call for papers announced the areas of interest for FOAL as including, but not limited to: formal semantics, formal specification, verification, theory of testing, aspect management, theory of aspect composition, and aspect translation and rewriting. The call for papers welcomed all theoretical and foundational studies of this topic.

The goals of this FOAL workshop were to:

- Explore the formal foundations of aspect-oriented programming.
- Exchange ideas about semantics and formal methods for aspect-oriented programming languages.
- Foster interest in the programming language theory communities concerning aspects and aspect-oriented programming languages.
- Foster interest in the formal methods community concerning aspects and aspect-oriented programming.

In addition, we hoped that the workshop would produce an outline of collaborative research topics and a list of areas for further exploration.

The papers at the workshop, which are included in the proceedings, were selected from papers submitted by researchers worldwide. Due to time limitations at the workshop, not all of the submitted papers were selected for presentation.

The workshop was organized by Ron Cytron (Washington University, St. Louis) and Gary T. Leavens (Iowa State University). The program committee that selected papers consisted of the organizers and James H. Andrews (U. Western Ontario), William Harrison (IBM T. J. Watson Research Center), K. Rustan M. Leino (Microsoft Research), Oscar Nierstrasz (U. of Berne), Wolfgang De Meuter (Vrije Universiteit Brussels), Jens Palsberg (Purdue Univ.), Kris De Volder (U. of British Columbia), and Mitch Wand (Northeastern University). We thank the organizers of AOSD 2002 for hosting the workshop.

A Semantics for Advice and Dynamic Join Points in Aspect-Oriented Programming

Mitchell Wand*
College of Computer Science
Northeastern University
360 Huntington Avenue, 161CN
Boston, MA 02115, USA
wand@ccs.neu.edu

Gregor Kiczales and Christopher Dutchyn
Department of Computer Science
University of British Columbia
201-2366 Main Mall
Vancouver, BC V6T 1Z4, Canada
{gregor,cdutchyn}@cs.ubc.ca

ABSTRACT

A characteristic of aspect-oriented programming, as embodied in AspectJ, is the use of *advice* to incrementally modify the behavior of a program. An advice declaration specifies an action to be taken whenever some condition arises during the execution of the program. The condition is specified by a formula called a *pointcut designator* or *pcd*. The events during execution at which advice may be triggered are called *join points*. In this model of aspect-oriented programming, join points are dynamic in that they refer to events during the execution of the program.

We give a denotational semantics for a minilanguage that embodies the key features of dynamic join points, pointcut designators, and advice. This is the first semantics for aspect-oriented programming that handles dynamic join points and recursive procedures. It is intended as a baseline semantics against which future correctness results may be measured.

1. INTRODUCTION

A characteristic of aspect-oriented programming, as embodied in AspectJ [11], is the use of *advice* to incrementally modify the behavior of a program. An advice declaration specifies an action to be taken whenever some condition arises during the execution of the program. The events at which advice may be triggered are called *join points*. In this model of aspect-oriented programming (AOP), join points are *dynamic* in that they refer to events during execution. The process of executing the relevant advice at each join point is called *weaving*.

The condition is specified by a formula called a *pointcut designator* or *pcd*. A typical pcd might look like

```
(and (pcalls f) (pwithin g) (cflow (pcalls h)))
```

This indicates that the piece of advice to which this pcd is attached is to be executed at every call to procedure `f` from within the text of procedure `g`, but only when that call occurs dynamically within a call to procedure `h`.

This paper presents a model of dynamic join points, pointcut designators, and advice. It introduces a tractable minilanguage embodying these features and gives it a denotational semantics.

This is the first semantics for aspect-oriented programming that handles dynamic join points and recursive procedures. It is intended as a baseline against which future correctness results may be measured.

This work is part of the Aspect Sandbox (ASB) project. The goal is of ASB to produce an experimental workbench for aspect-oriented programming of various flavors. ASB includes a small base language and is intended to include a set of exemplars of different approaches to AOP. The work reported here is a model of one of those exemplars, namely dynamic join points and advice with dynamic weaving. We hope to extend this work to other AOP models, including static join points, Demeter [14], and Hyper/J [16], and to both interpreter-like and compiler-like implementation models.

For more motivation for AOP, see [12] or the articles in [4]. For more on AspectJ, see [11].

2. A MODEL

We begin by presenting a conceptual model of aspect-oriented programming with dynamic join points as found in AspectJ.

In this model, a program consists of a base program and some pieces of *advice*. The program is executed by an interpreter. When the interpreter reaches certain points, called *join points*, in its execution, it invokes a *weaver*, passing to it an abstraction of its internal state (the *current join point*). Each advice contains a predicate, called a *pointcut designator* (*pcd*), describing the join points in which it is interested, and a body representing the action to take at those points. It is the job of the weaver to demultiplex the join points from the interpreter, invoking each piece of advice that is interested in the current join point and executing its body with the same interpreter.

*Work supported by the National Science Foundation under grant number CCR-9804115. An earlier version of this paper was presented at the 9th International Workshop on Foundations of Object-Oriented Languages, January 19, 2002.

So far, this sounds like an instance of the Observer pattern [8]. But there are several differences:

1. First, when a piece of advice is run, its body may be evaluated before, after or instead of the expression that triggered it; this specification is part of the advice. In the last case, called an *around* advice, the advice body may call the primitive `proceed` to invoke the running of any other applicable pieces of advice and the base expression.
2. Second, the language of predicates is a temporal logic, with temporal operators such as `cflow` illustrated above. Hence the current join point may in general be an abstraction of the control stack.
3. Each advice body is also interpreted by the same interpreter, so its execution may give rise to additional events and advice executions.
4. Last, in the language of this paper, as in the current implementation of AspectJ, the set of advice in each program is a global constant. This is in contrast with the Observer pattern, in which listeners register and de-register themselves dynamically.

This is of course a conceptual model and is intended only to motivate the semantics, not the implementation. However, this analysis highlights the major design decisions in any such language:

1. The join-point model: when does the interpreter call the weaver, and what data does it expose?
2. The pcd language: what is the language of predicates over join points? How is data from the join point communicated to the advice?
3. The advice model: how does advice modify the execution of the program?

In this paper, we explore one set of answers to these questions. Section 3 gives brief description of the language and some examples. Section 4 presents the semantics. In section 5 we describe some related work, and in section 6 we discuss our current research directions.

```
(run
  '((procedure void main ()
      (write (fact 3)))
    (procedure int fact ((int n))
      (if (< n 1) 1
        (* n (fact (- n 1)))))
    (around
      (and
        (pcalls int fact (int))
        (args (int x)))
      (let (((int y) 0))
        (write 'before1:)
        (write x) (newline)
        (set! y (proceed x))
        (write 'after1:)
        (write x) (write y) (newline)
        y))
    (around
      (and
        (pcalls int fact (int))
        (args (int x)))
      (let (((int y) 0))
        (write 'before2:) (write x)
        (newline)
        (set! y (proceed x))
        (write 'after2:)
        (write x) (write y) (newline)
        y))))
```

prints:

```
before1: 3
before2: 3
before1: 2
before2: 2
before1: 1
before2: 1
before1: 0
before2: 0
after2: 0 1
after1: 0 1
after2: 1 1
after1: 1 1
after2: 2 2
after1: 2 2
after2: 3 6
after1: 3 6
6
```

Figure 1: Example of around advice

3. EXAMPLES

Our base language consists of a set of mutually-recursive first-order procedures with a call-by-value interpretation. The language is first-order: procedures are not expressed values. The language includes assignment in the usual call-by-value fashion: new storage is allocated for every binding of a formal parameter, and identifiers in expressions are automatically dereferenced.

Figure 1 shows a simple program in this language, using the syntax of ASB. We have two pieces of `around` advice that are triggered by a call to `fact`.[1] At each advice execution, `x` will be bound to the argument of `fact`. The program begins by calling `main`, which in turn calls `fact`. The first advice body is triggered. Its body prints the `before1` message and then evaluates the `proceed` expression, which proceeds with the rest of the execution. The execution continues by invoking the second advice, which behaves similarly, printing the `before2` message; its evaluation of the `proceed` expression executes the actual procedure `fact`, which calls `fact` recursively, which invokes the advice again. Eventually `fact` returns 1, which is returned as the value of the `proceed` expression. As each `proceed` expression returns, the remainder of each advice body is evaluated, printing the various `after` messages.

Each `around` advice has complete control of the computation; further computation, including any other applicable advice, is undertaken only if the advice body calls `proceed`. For example, if the

[1] As shown in these examples, the executable version of ASB includes types for arguments and results. The portion of ASB captured by our semantics is untyped.

```
(run
  '((procedure void main ()
      (write (+ (fact 6) (foo 4))))
    (procedure int fact ((int n))
      (if (= n 0) 1
        (* n (fact (- n 1)))))
    (procedure int foo ((int n))
      (fact n))
    (before (and
              (pcalls int fact (int))
              (args (int y))
              (cflow
                (and
                  (pcalls int foo (int))
                  (args (int x)))))
      (write x) (write y) (newline))))
```

prints:

```
4 4
4 3
4 2
4 1
4 0
744
```

Figure 2: Binding variables with `cflow`

`proceed` in the first advice were omitted, the output would be just

```
before1: 3
after1: 3 0
0
```

The value of `x` must be passed to the `proceed`. If the call to `proceed` in the second advice were changed to `(proceed (- x 1))`, then `fact` would be called with "wrong" recursive argument. This design choice is intentional: changing the argument to `proceed` is a standard idiom in AspectJ.

Our language also includes `before` and `after` advice, which are evaluated on entry to and on exit from the join point that triggers them; these forms of advice do not require an explicit call to `proceed` and are always executed for effect, not value.

The language of pointcut designators includes temporal operators as well. Figure 2 shows an advice that is triggered by a call of `fact` that occurs within the dynamic scope of a call to `foo`. This program prints 720+24 = 744, but only the last four calls to `fact` (the ones during the call of `foo`) cause the advice to execute. The pointcut argument to `cflow` binds `x` to the argument of `foo`. Our language of pcd's includes several temporal operators. For example, `cflowtop` finds the oldest contained join point that satisfies its argument. Our semantics includes a formal model that explains this behavior.

The examples shown here are from the Aspect Sandbox (ASB). ASB consists of a base language, called BASE, and a separate language of advice and weaving, called AJD. The language BASE is a simple language of procedures, classes, and objects. Our intention is that the same base language be used with different weavers, representing different models of AOP; AJD is intended to capture the AspectJ dynamic join point style of AOP. The relation between AJD and BASE is intended to model the relationship between AspectJ and Java. We implemented the base language and AJD using an interpreter in Scheme in the style of [7].

For the semantics, we have simplified BASE and AJD still further by removing types, classes, and objects from the language and by slightly simplifying the join point model; the details are listed in the appendix. While much has been left out, the language of the semantics still models essential characteristics of AspectJ, including dynamic join points; pointcut designators; and `before`, `after`, and `around` advice.

4. SEMANTICS

We use a monadic semantics, using partial-function semantics whenever possible. In general, we use lower-case Roman letters to range over sets, and Greek letters to range over elements of partial orders.

Typical sets:

Sets

v	$\in$	*Val*	Expressed Values
l	$\in$	*Loc*	Locations
s	$\in$	*Sto*	Stores
id	$\in$	*Id*	Identifiers (program variables)
pname, *wname*	$\in$	*Pname*	procedure names

4.1 Join Points

We begin with the definition of join points. We use the term *join point* to refer both to the events during the execution of the program at which advice may run and to the portion of the program state that may be visible to the advice. The portion of the program state made visible to the advice consists of the following data:

Join points

jp	$\in$	*JP*	Join Points
jp	::=	$\langle\rangle \mid \langle k, pname, wname, v^*, jp\rangle$	
k	::=	`pcall` | `pexecution` | `aexecution`	
			Join Point Kinds

A join point is an abstraction of the control stack. It is either empty or consists of a kind, some data, and a previous join point. The join point $\langle$`pcall`, $f, g, v^*, jp\rangle$ represents a call to procedure f from procedure g, with arguments v^*, and with previous join point jp. `pexecution` and `aexecution` join points represent execution of a procedure or advice body; in these join points the three data fields contain empty values.

4.2 Pointcut Designators

A pointcut designator is a formula that specifies the set of join points to which a piece of advice is applicable. When applied to a

join point, a pointcut designator either succeeds with a set of bindings, or fails.

The grammar of pcd's is given by:

Pointcut designators

$$
\begin{aligned}
\mathit{pcd} &::= \texttt{(pcalls } \mathit{pname}\texttt{)} \mid \texttt{(pwithin } \mathit{pname}\texttt{)} \\
&::= \texttt{(args } \mathit{id}_1 \ \ldots \ \mathit{id}_n\texttt{)} \\
&::= \texttt{(and } \mathit{pcd}\ \mathit{pcd}\texttt{)} \mid \texttt{(or } \mathit{pcd}\ \mathit{pcd}\texttt{)} \mid \texttt{(not } \mathit{pcd}\texttt{)} \\
&::= \texttt{(cflow } \mathit{pcd}\texttt{)} \\
&::= \texttt{(cflowbelow } \mathit{pcd}\texttt{)} \mid \texttt{(cflowtop } \mathit{pcd}\texttt{)}
\end{aligned}
$$

The semantics of pcd's is given by a function *match-pcd* that takes a pcd and a join point and produces either a set of bindings (a finite partial map from identifiers to expressed values), or the singleton *Fail*.

Before defining *match-pcd*, we must define the operations on bindings and pcd results. We write $[]$ for the empty set of bindings and $+$ for concatenation of bindings. The behavior of repeated bindings under $+$ is unspecified. The operations $\vee$, $\wedge$, and $\neg$ on the result of *match-pcd* are defined by

Algebra of pcd results

$$
\begin{array}{lll}
b & \in & \mathit{Bnd} = [\mathit{Id} \to \mathit{Val}] \qquad \text{Bindings} \\
r & \in & \mathit{Optional\ (Bnd)} = \mathit{Bnd} + \{\mathit{Fail}\}
\end{array}
$$

$$
\begin{array}{lll}
b \vee r = b & \mathit{Fail} \wedge r = \mathit{Fail} & \neg \mathit{Fail} = [] \\
\mathit{Fail} \vee r = r & b \wedge \mathit{Fail} = \mathit{Fail} & \neg b = \mathit{Fail} \\
 & b \wedge b' = b + b' &
\end{array}
$$

Note that both $\wedge$ and $\vee$ are short-cutting, so that $\vee$ prefers its first argument.

We can now give the definition of *match-pcd*. *match-pcd* proceeds by structural induction on its first argument. The pcd's fall into three groups. The first group does pattern matching on the top portion of the join point: `(pcalls` *pname*`)` and `(pwithin` *pname*`)` check the target and within fields of the join point. `(args` $\mathit{id}_1 \ \ldots \ \mathit{id}_n$`)` succeeds if the argument list in the join point contains exactly n elements, and binds $\mathit{id}_1, \ldots, \mathit{id}_n$ to those values. In full AJD, the `args` pcd includes dynamic type checks as well.

*match-pcd***: basic operations**

$$
\begin{aligned}
&\mathit{match\text{-}pcd}\ \texttt{(pcalls } \mathit{pname}\texttt{)}\ \langle k, \mathit{pname}', \mathit{wname}, v^*, \mathit{jp}\rangle \\
&\quad = \begin{cases} [] & \text{if } k = \texttt{pcall} \wedge \mathit{pname} = \mathit{pname}' \\ \mathit{Fail} & \text{otherwise} \end{cases} \\
&\mathit{match\text{-}pcd}\ \texttt{(pwithin } \mathit{wname}\texttt{)}\ \langle k, \mathit{pname}, \mathit{wname}', v^*, \mathit{jp}\rangle \\
&\quad = \begin{cases} [] & \text{if } k = \texttt{pcall} \wedge \mathit{wname} = \mathit{wname}' \\ \mathit{Fail} & \text{otherwise} \end{cases} \\
&\mathit{match\text{-}pcd}\ \texttt{(args } \mathit{id}_1 \ \ldots \ \mathit{id}_n\texttt{)}\ \langle k, \mathit{pname}, \mathit{wname} \\
&\qquad\qquad\qquad\qquad (v_1, \ldots, v_m), \mathit{jp}\rangle \\
&\quad = \begin{cases} [\mathit{id}_1 = v_1, \ldots, \mathit{id}_n = v_n] & \text{if } k = \texttt{pcall} \text{ and } n = m \\ \mathit{Fail} & \text{otherwise} \end{cases}
\end{aligned}
$$

The second group, `(and` *pcd pcd*`)`, `(or` *pcd pcd*`)`, and `(not` *pcd*`)`, perform boolean combinations on the results of their arguments, using the functions $\wedge$, $\vee$, and $\neg$ defined above.

*match-pcd***: boolean operators**

$$
\begin{aligned}
&\mathit{match\text{-}pcd}\ \texttt{(and } \mathit{pcd}_1\ \mathit{pcd}_2\texttt{)}\ \mathit{jp} = \mathit{match\text{-}pcd}\ \mathit{pcd}_1\ \mathit{jp} \\
&\qquad\qquad\qquad\qquad \wedge \mathit{match\text{-}pcd}\ \mathit{pcd}_2\ \mathit{jp} \\
&\mathit{match\text{-}pcd}\ \texttt{(or } \mathit{pcd}_1\ \mathit{pcd}_2\texttt{)}\ \mathit{jp} = \mathit{match\text{-}pcd}\ \mathit{pcd}_1\ \mathit{jp} \\
&\qquad\qquad\qquad\qquad \vee \mathit{match\text{-}pcd}\ \mathit{pcd}_2\ \mathit{jp} \\
&\mathit{match\text{-}pcd}\ \texttt{(not } \mathit{pcd}\texttt{)}\ \mathit{jp} = \neg(\mathit{match\text{-}pcd}\ \mathit{pcd}\ \mathit{jp})
\end{aligned}
$$

Last, we have the temporal operators `(cflow` *pcd*`)`, `(cflowbelow` *pcd*`)`, and `(cflowtop` *pcd*`)`. The pcd `(cflow` *pcd*`)` finds the latest (most recent) join point that satisfies *pcd*. `(cflowbelow` *pcd*`)` is just like `(cflow` *pcd*`)`, but it skips the current join point, beginning its search at the first preceding join point; `(cflowtop` *pcd*`)` is like `(cflow` *pcd*`)`, but it finds the earliest matching join point. These searches can be thought of local loops within the overall structural induction.

*match-pcd***: temporal operators**

$$
\begin{aligned}
&\mathit{match\text{-}pcd}\ \texttt{(cflow } \mathit{pcd}\texttt{)}\ \langle\rangle = \mathit{Fail} \\
&\mathit{match\text{-}pcd}\ \texttt{(cflow } \mathit{pcd}\texttt{)}\ \langle k, \mathit{pname}, \mathit{wname}, v^*, \mathit{jp}\rangle \\
&\quad = \mathit{match\text{-}pcd}\ \mathit{pcd}\ \langle k, \mathit{pname}, \mathit{wname}, v^*, \mathit{jp}\rangle \\
&\qquad \vee \mathit{match\text{-}pcd}\ \texttt{(cflow } \mathit{pcd}\texttt{)}\ \mathit{jp} \\
&\mathit{match\text{-}pcd}\ \texttt{(cflowbelow } \mathit{pcd}\texttt{)}\ \langle\rangle = \mathit{Fail} \\
&\mathit{match\text{-}pcd}\ \texttt{(cflowbelow } \mathit{pcd}\texttt{)}\ \langle k, \mathit{pname}, \mathit{wname}, v^*, \mathit{jp}\rangle \\
&\quad = \mathit{match\text{-}pcd}\ \texttt{(cflow } \mathit{pcd}\texttt{)}\ \mathit{jp} \\
&\mathit{match\text{-}pcd}\ \texttt{(cflowtop } \mathit{pcd}\texttt{)}\ \langle\rangle = \mathit{Fail} \\
&\mathit{match\text{-}pcd}\ \texttt{(cflowtop } \mathit{pcd}\texttt{)}\ \langle k, \mathit{pname}, \mathit{wname}, v^*, \mathit{jp}\rangle \\
&\quad = \mathit{match\text{-}pcd}\ \texttt{(cflowtop } \mathit{pcd}\texttt{)}\ \mathit{jp} \\
&\qquad \vee \mathit{match\text{-}pcd}\ \mathit{pcd}\ \langle k, \mathit{pname}, \mathit{wname}, v^*, \mathit{jp}\rangle
\end{aligned}
$$

4.3 The Execution Monad

To package the execution, we introduce a monad:

$$T(A) = JP \to Sto \to (A \times Sto)_\perp$$

This is a monad with three effects: a dynamically-scoped quantity of type *JP*, a store of type *Sto*, and non-termination. It says that a computation runs given a join point and a store, and either produces a value and a store, or else fails to terminate. The monad operations ensure that *JP* has dynamic scope and that *Sto* is global:

Monad operations

$$\begin{array}{l}\textbf{return}\ v = \lambda\, jp\, s\,.\, lift(v,s)\\ \textbf{let}\ v \Leftarrow E_1\ \textbf{in}\ E_2\\ \quad = \lambda\, jp\, s\,.\, \textbf{case}\ (E_1\ jp\ s)\ \textbf{of}\\ \qquad\qquad \perp \Rightarrow \perp\\ \qquad\qquad lift(v,s') \Rightarrow ((\lambda v\,.\,E_2)\ v\ jp\ s')\end{array}$$

We write

$$\textbf{let}\ v_1 \Leftarrow \mu_1; \ldots; v_n \Leftarrow \mu_n\ \textbf{in}\ E$$

for the evident nested **let**.

We will have the usual monadic operations on the store; for join points we will have a single monadic operator **setjp**. **setjp** takes a function f from join points to join points and a map g from join points to computations. It returns a computation that applies f to the current join point, passes the new join point to g, and runs the resulting computation in the new join point and current store:

setjp

$$\begin{array}{l}\textbf{setjp} : (JP \to JP) \to (JP \to T(A)) \to T(A)\\ \quad = \lambda\, f\ g\,.\,\lambda\, jp\, s\,.\,(g\ (f\ jp))\ (f\ jp)\ s\end{array}$$

The *lift* operation induces an order on $T(A)$ for any A. We will use the following domains based on this order:

Domains

χ	$\in$	$T(Val)$	Computations
π	$\in$	$Proc = Val^* \to T(Val)$	Procedures
α	$\in$	$Adv = JP \to Proc \to Proc$	Advice
ϕ	$\in$	$PE = Pname \to Proc$	Procedure Environments
γ	$\in$	$AE = Adv^*$	Advice Environments
ρ	$\in$	$Env = [Id \to Loc] \times WName \times Proceed$	Environments
		$WName = Optional(Pname)$	Within Info
		$Proceed = Optional(Proc)$	`proceed` Info

A procedure takes a sequence of arguments and produces a computation. An advice takes a join point and a procedure, and produces a new procedure that is either the original procedure wrapped in the advice (if the advice is applicable at this join point) or else is the original procedure unchanged (if the advice is inapplicable). Procedures and advice do not require any environment arguments because they are always defined globally and are closed (mutually recursively) in the global procedure- and advice- environments.

The distinguished *WName* component of the environment will be used for tracking the name of the procedure (if any) in which the current program text resides. Similarly, the distinguished *Proceed* component will be used for the `proceed` operation, if it is defined. We write $\rho(\texttt{\%within})$, $\rho[\texttt{\%within} = \ldots]$, $\rho(\texttt{\%proceed})$, and $\rho[\texttt{\%proceed} = \ldots]$ to manipulate these components.

4.4 Expressions

We can now give the semantics of expressions. We give here only a fragment:

Semantics of expressions

$$\begin{array}{l}\mathcal{E}[\![e]\!] \in Env \to PE \to AE \to T(Val)\\ \\ \mathcal{E}[\![(pname\ \ e_1\ \ \ldots\ \ e_n)]\!]\rho\phi\gamma\\ \quad = \ \textbf{let}\ v_1 \Leftarrow \mathcal{E}[\![e_1]\!]\rho\phi\gamma;\ \ldots;\ v_n \Leftarrow \mathcal{E}[\![e_n]\!]\rho\phi\gamma\\ \qquad \textbf{in}\ (\textit{enter-join-point}\ \gamma\\ \qquad\quad (\textit{new-pcall-jp}\ pname\ (\rho\ \texttt{\%within})\ (v_1,\ldots,v_n))\\ \qquad\quad (\phi\,(pname))\\ \qquad\quad (v_1,\ldots,v_n))\\ \\ \mathcal{E}[\![(\texttt{proceed}\ \ e_1\ \ \ldots\ \ e_n)]\!]\rho\phi\gamma\\ \quad = \textbf{let}\ v_1 \Leftarrow \mathcal{E}[\![e_1]\!]\rho\phi\gamma;\ \ldots;\ v_n \Leftarrow \mathcal{E}[\![e_n]\!]\rho\phi\gamma\\ \qquad \textbf{in}\ \rho\,(\texttt{\%proceed})\,(v_1,\ldots,v_n)\end{array}$$

In a procedure call, first the arguments are evaluated in the usual call-by-value monadic way. Then, instead of directly calling the procedure, we use *enter-join-point* to create a new join point and enter it, invoking the weaver to apply any relevant advice. Contrast this with the `proceed` expression, which is like a procedure call, except that the special procedure `%proceed` is called, and no additional weaving takes place. The function *new-pcall-jp* : $Pname \to WName \to Val^* \to JP \to JP$ builds a new procedure-call join point following the grammar in section 4.1.

4.5 The Weaver and Advice

enter-join-point is the standard entry to a new join point. It takes a list of advice γ, a join-point builder f, a procedure π, and a list of arguments v^*. It produces a computation that builds a new join point using function f, calls the weaver to wrap all the advice in γ around procedure π, and then applies the resulting procedure to v^*.

enter-join-point

$$\begin{aligned}&\textit{enter-join-point} : \textit{AE} \to (\textit{JP} \to \textit{JP}) \to \textit{Proc} \to \textit{Proc}\\ &\quad = \lambda\,\gamma\ f\ \pi.\lambda\,v^*\,.\ \textbf{setjp}\ f\ (\lambda\,jp'\,.\ \textit{weave}\ \gamma\ jp'\ \pi\ v^*)\end{aligned}$$

The weaver is the heart of the system. It takes a list of advice, a join point, and a procedure. It returns a new procedure that consists of the original procedure wrapped in all of the advice that is applicable at the join point. To do this, the weaver attempts to apply each piece of advice in turn. If there is no advice left, then the effective procedure is just the original procedure π. Otherwise, it calls the first advice in the list, asking it to wrap its advice (if applicable) around the procedure that results from weaving the rest of the advice around the original procedure.

So we want

$$(\textit{weave}\ \langle\alpha_1,\ldots,\alpha_n\rangle\ jp\ \pi) = (\alpha_1\ jp\ (\alpha_2\ jp\ \ldots\ (\alpha_n\ jp\ \pi)\ldots))$$

This becomes a straightforward bit of functional programming:

The weaver

$$\begin{aligned}&\textit{weave} : \textit{AE} \to \textit{JP} \to \textit{Proc} \to \textit{Proc}\\ &\quad = \lambda\,\gamma\ jp\ \pi\ .\textbf{case}\ \gamma\ \textbf{of}\\ &\qquad\qquad \langle\rangle \Rightarrow \pi\\ &\qquad\qquad \alpha :: \gamma' \Rightarrow \alpha\ jp\ (\textit{weave}\ \gamma'\ jp\ \pi)\end{aligned}$$

This brings us to the semantics of advice. A piece of advice, like an expression, should take a procedure environment and an advice environment, and its meaning should be a procedure transformer. Our fundamental model is `around` advice. If the advice does not apply in the current join point, then the procedure should be unchanged. If the advice does apply, then the advice body should be executed with the bindings derived from the pcd, and with `%proceed` set to the original procedure (which may be either the starting procedure or a procedure containing the rest of the woven advice). However, there are two subtleties: first, the body of the advice is to be executed in a new `aexecution` join point, so we use *enter-join-point* to build the new join point and invoke the weaver. This is potentially an infinite regress, so most advice pcd's will include an explicit `pcalls` conjunct to avoid this problem. Second, in this case, the inner v^* is not used; the advice body can retrieve it using an `args` pcd.

`before` and `after` advice are similar; `%proceed` is not bound, and we use the monad operations to perform the sequencing.

Semantics of advice

$$\begin{aligned}&\mathcal{A}[\![\texttt{(around}\ \textit{pcd}\ \ \textit{e)}]\!]\phi\gamma : \textit{JP} \to \textit{Proc} \to \textit{Proc}\\ &\quad = \lambda\,jp\,\pi\,v^*\,.\\ &\qquad \mathcal{PCD}[\![\textit{pcd}]\!]\,jp\\ &\qquad\quad (\lambda\rho.\,\textit{enter-join-point}\ \gamma\\ &\qquad\qquad \textit{new-aexecution-jp}\\ &\qquad\qquad (\lambda\,v^*\,.\,\mathcal{E}[\![e]\!](\rho[\texttt{\%within} = \textit{None},\\ &\qquad\qquad\qquad\qquad \texttt{\%proceed} = \pi]\phi\gamma))\\ &\qquad\qquad \langle\rangle)\\ &\qquad\quad (\pi\ v^*)\end{aligned}$$

$$\begin{aligned}&\mathcal{A}[\![\texttt{((before}\ \textit{pcd)}\ \ \textit{e)}]\!]\phi\gamma : \textit{JP} \to \textit{Proc} \to \textit{Proc}\\ &\quad = \lambda\,jp\,\pi\,v^*\,.\\ &\qquad \mathcal{PCD}[\![\textit{pcd}]\!]\,jp\\ &\qquad\quad (\lambda\rho.\,\textit{enter-join-point}\ \gamma\\ &\qquad\qquad \textit{new-aexecution-jp}\\ &\qquad\qquad (\lambda\,v^*\,.\ \textbf{let}\\ &\qquad\qquad\qquad v_1 \Leftarrow \mathcal{E}[\![e]\!](\rho[\texttt{\%within} = \textit{None},\\ &\qquad\qquad\qquad\qquad\qquad \texttt{\%proceed} = \textit{None}])\phi\gamma;\\ &\qquad\qquad\qquad v_2 \Leftarrow (\pi\ v^*)\\ &\qquad\qquad\qquad \textbf{in}\ v_2)\\ &\qquad\qquad \langle\rangle)\\ &\qquad\quad (\pi\ v^*)\end{aligned}$$

$$\begin{aligned}&\mathcal{A}[\![\texttt{((after}\ \textit{pcd)}\ \ \textit{e)}]\!]\phi\gamma : \textit{JP} \to \textit{Proc} \to \textit{Proc}\\ &\quad = \lambda\,jp\,\pi\,v^*\,.\\ &\qquad \mathcal{PCD}[\![\textit{pcd}]\!]\,jp\\ &\qquad\quad (\lambda\rho.\,\textit{enter-join-point}\ \gamma\\ &\qquad\qquad \textit{new-aexecution-jp}\\ &\qquad\qquad (\lambda\,v^*\,.\ \textbf{let}\\ &\qquad\qquad\qquad v_1 \Leftarrow (\pi\ v^*);\\ &\qquad\qquad\qquad v_2 \Leftarrow \mathcal{E}[\![e]\!](\rho[\texttt{\%within} = \textit{None},\\ &\qquad\qquad\qquad\qquad\qquad \texttt{\%proceed} = \textit{None}])\phi\gamma\\ &\qquad\qquad\qquad \textbf{in}\ v_1)\\ &\qquad\qquad \langle\rangle)\\ &\qquad\quad (\pi\ v^*)\end{aligned}$$

The function $\mathcal{PCD}[\![-]\!]$ takes four arguments: a pcd, a join point, a function k from environments to computations (the "success continuation"), and a computation χ (the "failure computation"), and it produces a computation. It calls *match-pcd* to match the pcd against the join point. If *match-pcd* succeeds with a set of bindings, $\mathcal{PCD}$ creates an environment containing a fresh location for each binding, and invokes the success continuation on this environment, producing a new computation. Otherwise, it returns the failure computation.

Semantics of pcd's

$$\begin{aligned}&\mathcal{PCD}[\![pcd]\!] : JP \to (Env \to T(Val)) \to T(Val) \to T(Val)\\ &\quad = \lambda\, jp\; k\chi.\, \textbf{case}\,(\textit{match-pcd pcd jp})\,\textbf{of}\\ &\qquad\qquad \textit{Fail} \Rightarrow \chi\\ &\qquad\qquad [x_1 = v_1, \ldots, x_n = v_n] \Rightarrow\\ &\qquad\qquad\quad \textbf{let}\; l_1 \Leftarrow \textbf{alloc}(v_1); \ldots; l_n \Leftarrow \textbf{alloc}(v_n)\\ &\qquad\qquad\quad \textbf{in}\; k([x_1 = l_1, \ldots, x_n = l_n])\end{aligned}$$

4.6 Procedures and Programs

Finally, we give the semantics of procedures and whole programs. The meaning of a procedure in a procedure and advice environment is a small procedure environment. In this environment, the name of the procedure is bound to a procedure that accepts some arguments and enters a `pexecution` join point, possibly weaving some advice. When the advice is accounted for, the arguments are stored in new locations, and the procedure body is executed in an environment in which the formal parameters are bound to the new locations.

Semantics of procedure declarations

$$\begin{aligned}&\mathcal{P}[\![\texttt{(procedure}\ \textit{pname}\ \texttt{(}x_1\ \ldots\ x_n\texttt{)}\ e\texttt{)}]\!] : PE \to AE \to PE\\ &\quad = \lambda\phi\gamma.[\textit{pname} =\\ &\qquad\quad \lambda v^*.(\textit{enter-join-point}\ \gamma\\ &\qquad\qquad\quad (\textit{new-pexecution-jp pname})\\ &\qquad\qquad\quad (\lambda w.\ \ \textbf{let}\, l_1 \Leftarrow \textbf{alloc}(w{\downarrow}1)\,;\\ &\qquad\qquad\qquad\qquad \vdots\\ &\qquad\qquad\qquad\qquad l_n \Leftarrow \textbf{alloc}(w{\downarrow}n)\\ &\qquad\qquad\qquad\quad \textbf{in}\,(\mathcal{E}[\![e]\!][x_1 = l_1,\ \ldots,\ x_n = l_n,\\ &\qquad\qquad\qquad\qquad\quad \%\texttt{within} = \textit{pname},\\ &\qquad\qquad\qquad\qquad\quad \%\texttt{proceed} = \textit{None}]\ \phi\ \gamma))\\ &\qquad\qquad\quad v^*)]\end{aligned}$$

We have formulated the semantics of procedures and advice as being closed in a given procedure environment and advice environment. A program is a mutually recursive set of procedures and advice, so its semantics is given by the fixed point over these functions. We take the fixed point and then apply the procedure `main` to no arguments.

Semantics of programs

$$\begin{aligned}&\mathcal{PGM}[\![\texttt{(}\textit{proc}_1\ \ldots\ \textit{proc}_n\ \textit{adv}_1 \ldots\ \textit{adv}_m\texttt{)}]\!] : T(Val)\\ &\quad = \textit{run}(\textit{fix}(\lambda(\phi,\gamma).(\textstyle\sum_{i=1}^{n}(\mathcal{P}[\![\textit{proc}_i]\!]\phi\gamma), \langle\mathcal{A}[\![\textit{adv}_j]\!]\phi\gamma\rangle_{j=1}^{m})))\\ &\textit{run}(\phi,\gamma) = \mathcal{E}[\![\texttt{(main)}]\!][]\phi\gamma\end{aligned}$$

Here the notation $\langle\ldots\rangle_{j=1}^{m}$ denotes a sequence of length m, and the notation $\sum_{i=1}^{n}$ denotes the concatenation operator on bindings, discussed on page .

This completes the semantics of the core language.

5. RELATED WORK

Aspect-oriented programming is presented in [12], which shows how several elements of prior work, including reflection [17], metaobject protocols [10], subject-oriented programming [9], adaptive programming [14], and composition filters [1] all enable better control over modularization of crosscutting concerns. A variety of models of AOP are presented in [4]. AspectJ [11] is an effort to develop a Java-based language explicitly driven by the principles of AOP.

Flavors [19, 5], New Flavors [15], CommonLoops [3] and CLOS [18] all support `before`, `after`, and `around` methods.

Andrews [2] presents a semantics for AOP programs based on a CSP formalism, using CSP synchronization sets as join points. His language is an imperative language with first-order procedures, like ours, but it does not allow procedures to be recursive. His language includes `before`, `after`, and `around` advice, but his pcd's contain neither boolean nor temporal operators.

Lämmel [13] presents static and dynamic operational semantics for a small OO language with a method-call interception facility somewhat different from ours. His system allows dynamic registration of advice, but does not treat `around` advice.

Douence, Motelet, and Sudholt [6] present an event-based theory of AOP. They present a domain-specific language for defining "crosscuts" (equivalent to our pointcuts). Their language is very powerful, but its semantics is given by a rewriting semantics, which makes the meaning of its programs obscure. We believe that our definition of *match-pcd* represents a significant improvement.

6. FUTURE WORK

We are currently developing a translator from AJD(BASE) to BASE that removes all advice by internalizing the weaving process. We hope to do this in a way that will facilitate a correctness proof.

We plan to extend the ASB suite by adding implementations of the core concepts of other models of AOP and weaving, including static join points, Demeter [14], and Hyper/J [16]. We hope to develop a theory of AOP that accounts for all of these.

7. REFERENCES

[1] M. Aksit, K. Wakita, J. Bosch, L. Bergmans, and A. Yonezawa. Abstracting Object Interactions Using Composition Filters. In R. Guerraoui, O. Nierstrasz, and M. Riveill, editors, *Proceedings of the ECOOP'93 Workshop on Object-Based Distributed Programming*, LNCS 791, pages 152–184. Springer-Verlag, 1994.

[2] J. H. Andrews. Process-algebraic foundations of aspect-oriented programming. In *Proceedings of the Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns (Reflection 2001)*, volume 2192 of *Lecture Notes in Computer Science*, pages 187–209, Berlin, Heidelberg, and New York, Sept. 2001. Springer-Verlag.

[3] D. G. Bobrow, K. Kahn, G. Kiczales, L. Masinter, M. Stefik, and F. Zdybel. CommonLoops: merging Common Lisp and object-oriented programming. In *Proceedings ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 17–29, Oct. 1986.

[4] *Communications of the ACM*, volume 44:10. ACM, Oct. 2001. special issue on Aspect-Oriented Programming.

[5] H. I. Cannon. *Flavors: A non-hierarchical approach to object-oriented programming*. Symbolics, Inc., 1982.

[6] R. Douence, O. Motelet, and M. Sudholt. A formal definition of crosscuts. In *Proceedings of the Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns (Reflection 2001)*, volume 2192 of *Lecture Notes in Computer Science*, pages 170–186, Berlin, Heidelberg, and New York, Sept. 2001. Springer-Verlag.

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

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

[9] W. Harrison and H. Ossher. Subject-oriented programming (A critique of pure objects). In A. Paepcke, editor, *Proceedings ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 411–428. ACM Press, Oct. 1993.

[10] G. Kiczales and J. des Rivieres. *The art of the metaobject protocol*. MIT Press, Cambridge, MA, USA, 1991.

[11] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersen, J. Palm, and W. G. Griswold. An overview of AspectJ. In *Proceedings European Conference on Object-Oriented Programming*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–353, Berlin, Heidelberg, and New York, 2001. Springer-Verlag.

[12] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Akşit and S. Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.

[13] R. Lämmel. A semantical approach to method-call interception. In G. Kiczales, editor, *1st International Conference on Aspect-Oriented Software Development*, Apr. 2002.

[14] K. J. Lieberherr. *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*. PWS Publishing Company, 1996.

[15] D. A. Moon. Object-oriented programming with Flavors. In N. Meyrowitz, editor, *Proceedings ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 1–8, New York, NY, Nov. 1986. ACM Press.

[16] H. Ossher and P. Tarr. Hyper/J: multi-dimensional separation of concerns for Java. In *Proceedings of the 22nd International Conference on Software Engineering, June 4-11, 2000, Limerick, Ireland*, pages 734–737, 2000.

[17] B. C. Smith. Reflection and semantics in Lisp. In *Conf. Rec. 11th ACM Symposium on Principles of Programming Languages*, pages 23–35, 1984.

[18] G. L. Steele. *Common Lisp: the Language*. Digital Press, Burlington MA, second edition, 1990.

[19] D. Weinreb and D. A. Moon. Flavors: Message passing in the LISP machine. A. I. Memo 602, Massachusetts Institute of Technology, A.I. Lab., Cambridge, Massachusetts, 1981.

APPENDIX A. LANGUAGE COMPARISON

Full AJD contains the following features not in the core language captured by the semantics of this paper. None represent difficult extensions for the semantics.

- classes, methods, and objects.
- declared types for bound variables (as illustrated in the examples of section 3).
- static type checking (an `args` pcd includes types for its arguments, as in our examples; at present these must be checked dynamically).
- additional join points at: method calls, method executions, object constructions, field references and field assignments.
- The pcd operators `and` and `or` take an arbitrary number of arguments.

AspectJ provides a sophisticated advice ordering mechanism, where advice is first ordered from most general to most specific, and within classes with equal specificity, orders the advice by qualifier (`before`, `after`, or `around`). AJD is working toward this capability, but the current stable implementation only provides the qualifier-based ordering, where `around` advice is executed around any relevant `before` and `after` advice. In the semantics, advice is ordered by its appearance in the program text.

The examples of section 3 were in written and executed in full AJD except for the following:

- the output was edited to improve formatting
- in the implementation of ASB at the time this work was done, eligible `around` advice was executed in reverse order from its appearance in the program text. The example in figure 1 was edited, reversing the order of advice declarations, to be consistent with the left-to-right semantics of the core language.

Member-Group Relationships Among Objects

William Harrison, Harold Ossher
IBM T. J. Watson Research
P.O. Box 704
Yorktown Heights, NY 10598
{harrisn, ossher}@watson.ibm.com

ABSTRACT

Aspect-Oriented Software is a broad term, encompassing several different views on the nature of the aspects and the relationships between aspects and objects. Attaching aspects to objects is one way of forming a group. While there are many useful patterns of interaction, e.g. strategies [2], decorators, and the like, we focus on groups in which the group delegates to members to obtain behavior and the members may either perform their own behavior or delegate to the group. Using issues of behavior, this paper explores and classifies the relationships between objects and groups of objects in which they may participate as a first step in laying a foundation for unifying these different views as special cases of a common framework.

Keywords

Aspect-oriented software development, delegation, composition, method combination.

1. INTRODUCTION

Different mechanisms in the AOSD space emphasize different means of separating and integrating concerns. For example, Hyper/J [9] focuses on composing class hierarchies, which in turn involves synthesizing composed classes and their methods from input classes. AspectJ [5] focuses attaching *advice* and *aspects* to *join points* and objects. The composition filters approach [1] focuses on attaching *filters* to objects to filter method calls and returns. These approaches all allow attachment of additional behavior to objects and/or combination of objects to form single objects with combined behavior.

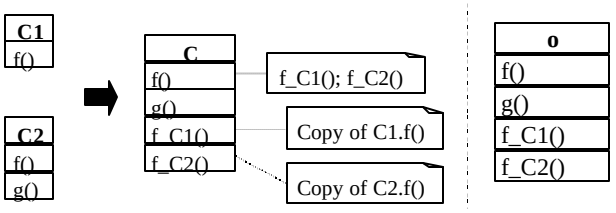

Figure 1. Hyper/J Class Composition and Example Instance

A simple example of class composition is shown in **Figure 1**. Input classes C1 and C2 are composed to form class C. All instantiations of classes C1 and C2 are changed to instantiations of C, so that, at runtime, only C instances exist, like *o* in the figure.

Earlier work on tool integration, including event broadcasting [11], cooperative call [8], and *mediators* [12], on the other hand, was concerned with tying together sets of separate objects (or other modules, but we confine ourselves to objects in this paper). Mediators, for example, provide for *implicit invocation*, so that events in an object can trigger a mediator, which can then trigger actions in other objects. AOSD approaches like those mentioned above do provide implicit invocation but do not address the implicit binding together of behaviors of several base objects performed by tool integration mechanisms.

Object-oriented programmers often split the implementation of functionality across several objects, relying on them to cooperate in carefully-designed ways to achieve the desired objective [4]. A common approach is *delegation*, in which part of the behavior of an object is specified in and delegated to other objects (or classes), such as *strategy* objects [2]. In conversations about Hyper/J, several developers have told us that they would prefer a model where composition produced object collaborations rather than single, composed objects.

Separating functionality into separate objects also provides more dynamic flexibility. Provided great care is exercised to coordinate the activity among all affected objects, it is possible to dynamically add objects to or remove them from a group, thereby adding or removing functionality, or replacing implementations. This kind of dynamism has always been important in some contexts, such as long-running telephone switches, and is becoming ever more important in the context of web-based applications.

Serious problems can arise, however, when the functionality that conceptually belongs in a single object is split across multiple objects in a group. These include *object schizophrenia* and *broken delegation* [14], and are due to the fact that the separate objects making up the group have their own, separate identities; even if they cooperate, they don't truly behave like a single object unless great care is taken, and the breakages are subtle. This paper analyzes these issues, and discusses various ways of handling identity and the relationships between objects and their groups, and their implications.

The first section analyzes a number of the major factors that characterize the ways in which an object's behavior can be related to a group of which it is part and then applies these factors to enumerating the potential kinds of relationships between objects and their groups. We winnow the enumeration by analyzing conflicts and usages that can lead to difficulties. The section concludes with a discussion of synergy and conflict in the relationships a single object may bear to multiple groups. The second section builds on and re-applies the factor analysis and winnowing process to

classes instead of instances of objects. The third section discusses the behavior of composition operations, responsible for aggregating objects into groups in order to realize composed behavior, when dealing with potential conflicts in multiple relationships. The fourth section discusses some options for implementing groups of objects and the composition operations that perform them, and analyzes the implications of some implementation decisions.

2. INSTANCE RELATIONSHIPS

2.1 Groups of Primitive Objects

Assume that, in concept, Java™ objects are either *primitive objects*, with fields and method bodies written by developers, or *group objects*, representing collections of objects that have been composed together. Group objects are created by composition operations, and do nothing but call methods of primitive objects (or, perhaps, of other groups) as determined by the compositions. Assume also that the group exposes the interfaces of all its member objects, so that methods of the primitive objects in a group can also be called on the group object itself. The group will generally delegate such operations to the appropriate primitive object(s), including other objects providing advice, filters, or additional implementations. The group thus serves as a kind of method combination dispatcher, determining how the composed behavior of each method call is to be realized in terms of the primitive methods supplied by the group members.

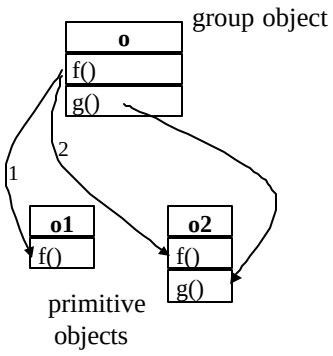

Figure 2. A Simple Group

Figure 2 shows a group of objects that realizes the same composition as illustrated in **Figure 1**. The entire group corresponds to the single, combined instance, *o*, in **Figure 1**. In this case, however, separate instances o1 and o2 of the original, uncomposed classes C1 and C2 do exist at runtime, as well as the group object, *o*, that ties them together.

Within the bodies of primitive object methods, *either of two* identities might be used, called *this* and *self*.[1] Many factors enter the analysis, but when they are different, *this* refers to the primitive and *self* to the group. Calls directed to *this* are thus directed to the primitive object itself, and do not invoke composed behavior, whereas calls to *self* are directed to the caller's group, and do invoke composed behavior.

[1] These terms are not ideal, because they are typically used to mean the same concept (object self-reference), though in different languages. Here we are using them to denote different concepts within the same model. We are in the process of trying to come up with better terms.

2.2 Factors in Describing Relationships

Trying to remain independent of the way the behaviors are actually implemented, we now explore and categorize the kinds of relationships among primitive and group objects to lay the groundwork for systematic support.

Leaving aside, until Section 2.4, situations in which groups act as members of larger groups, each kind of relationship between a primitive object and a group can be operationally characterized by several effects. The following table lists the relationships along with the effects ascribed to each. Explanation of columns:

Identity	Assuming that Java's reference equality semantics are appropriately extended, comparison of the identity of a primitive object and the identity of its group object can yield "equal" or "not equal".
Primitive-to-group	When a primitive object calls a method on a primitive object, the primitive object can cause group behavior rather than use its own method implementation. Three alternatives can be listed (see Figure 3):
	no — The primitive does not cause group behavior, but performs its primitive behavior instead.
	identical — The primitive yields to common group behavior (that which results when the method is called on the group object)
	variant — The primitive causes group behavior different from the common group behavior (such as including its own behavior in addition to the common group behavior).
Group-to-primitive	**When a method is called on a group, the group uses behavior defined by various primitive objects of the group. Three alternatives can be listed for how the group uses the primitive's behavior (see** Figure 4):
	no — The primitive's behavior is not included in the group behavior.
	self=primitive — Group behavior includes the primitive's behavior, but in interpreting the primitive's behavior, references to itself as *self*, are not to be interpreted

as if referring to the group, but as references to the primitive.

self= group — Group behavior includes the primitive's behavior and in interpreting the primitive's behavior, references to itself as *self* are to be interpreted as references to the group rather than as references to the primitive. (But see the automanipulation alternatives, next).

Auto-manipulation — When *self*=group, there are a number of ways in which the behavior in a Java method may refer to the group by using *self*. The developer might explicitly refer to *self*, if this is permitted. (It would be, in effect, a Java language extension whose normal Java semantics might be innocuous.) Without a language extension, reference to the group can arise by reinterpreting the Java "*this*". Three cases can be listed (see Figure 5):

this= primitive — Explicit and implicit uses of *this* refer to the primitive.

this= group — Explicit and implicit uses of *this* use the value of *self*, and in this case, *self*=group.

mixed — Although there are hundreds of different mixed variations, corresponding to the different ways in which *this* appears in Java the most frequent suggestion is to make manifest uses of *this* refer to the primitive while other uses use the value of *self*. The only reason that this variation is particularly interesting is that, accomplished in-spite-of any general policy, it can be forced by a developer who copies the bodies of *final* methods into places where they are manifestly invoked on *this*.

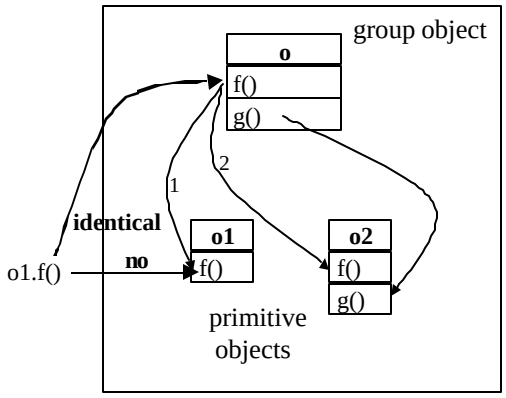

Figure 3. Primitive-to-Group Options "no" and "identical"

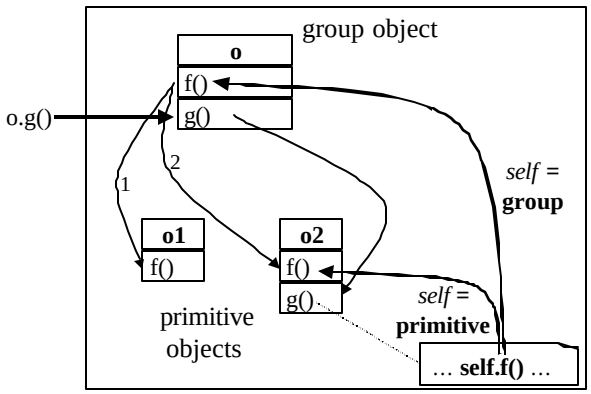

Figure 4. Group-to-Primitive Options "self = primitive" and "self = group"

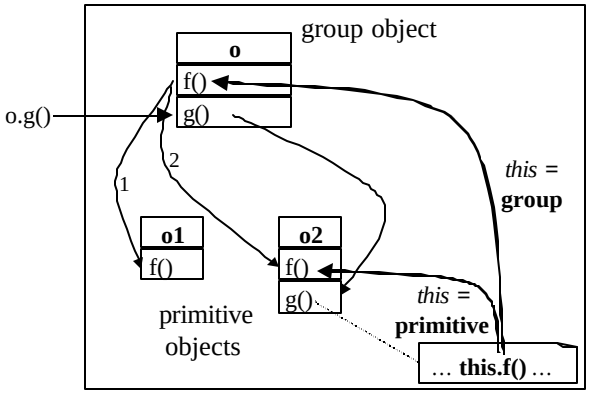

Figure 5. Automanipulation Options "this = primitive" and "this = group"

2.3 Relationships Induced By the Factors

We can make some general observations that reduce the resulting number of enumerable forms to 7, with what we believe to be less controversial rules first:

- Group-to-primitive forms of "no" or "*self*=primitive" render the automanipulation choice irrelevant, eliminating 24 of the 54 enumerable relationships.
- When the primitive-to-group behavior form is "identical", the automanipulation forms of "*this*=primitive" and "*this*=group" are equivalent. This rules out one of the remaining enumerable relationships.
- One of the most useful operational definitions of identity is that two objects have the same identity iff performing an operation on one of them always has the same result as performing the operation on the other. Of the remaining 29 enumerable relationships, this "identity rule" rules out the 10 in which "identity" disagrees with "primitive-to-group".
- Mixed "automanipulation" can be made only with respect to the coding of the body of an object's methods. This is an invasive, coding-dependent process that is probably better carried out by a developer making explicit use of *self* and adopting the *"this*=primitive" form of automanipulation. On the grounds of this fragility, we believe that manifest and other mixed automanipulation forms should be avoided and that the relationships should be deprecated. Of the 19 remaining enumerable relationships, "'mixed' deprecated" rules out 4.
- The variant form of primitive-to-group interaction can lead to a rather confusing collection of behaviors in which each member of the group has different behavior. We are left to wonder why this construct should be regarded as a group at all. An alternative would be to treat each varied behavior as a group of its own, with which the members can be associated on an "identical" footing. Of the 15 remaining enumerable relationships, "'variant' deprecated" encompasses 8.
- We have reservations about relationship 6 (maverick). The claim is that the object has the group's identity and has group behavior when called from primitive objects, but when called from the group its self-calls are not given group behavior. But we do not see a contradiction or a reasonable rule of meaning that prohibits the relationship.

Name for object's relationship to group	Identity	Primitive-to-group	Group-to-primitive	Automanipulation
1. Stand-alone	unequal	no	no	*(this=self=primitive)*
2. Associate	unequal	no	*self*=primitive	*(this=self=primitive)*
3. Aspect	unequal	no	*self*=group	*this*=primitive
4. Affiliate	equal		group	*this*=group
("mixed" deprecated)				mixed
("variant" deprecated)	unequal	variant	—	—
(Violates identity rule)	unequal	identical	—	—
equivalent to 5	equal	identical	*self*=group	*this*=primitive
5. Facet				*this*=group
("mixed" deprecated)				mixed
6. Maverick	equal	identical	*self*=primitive	*(this=self=primitive)*
7. Router	equal	identical	no	*(not used)*
("variant" deprecated)	equal	variant	—	—
(Violates identity rule)	equal	no	—	—

We could, of course, rule out any of these relationships for implementation convenience.

2.4 Objects in Multiple Relationships

We can also examine the question of what relationships an object can have simultaneously to two groups.

	1	2	3	4	5	6	7	Notes
1		Y	Y	Y	N	N	N	Stand-alone doesn't pass control to a group when called from outside
2		Y	Y	Y	Y	Y	Y	Associate, Aspect, Affiliate can coexist with being stand-alone or with being in a group.
3			Y	Y	Y	Y	Y	
4				Y	Y	Y	Y	
5					N	N	N	Can delegate to at most one group when called from outside
6						N	N	
7							N	

2.5 Higher-Order (Group-Group) Relationships

Allowing groups to be members of other groups introduces no new situations. For the nonce, call the group with groups as members a "supergroup", although we intend to observe that it is no different from any other group. Since all real method function lies in primitives, a supergroup never need use a group as an intermediary. With appropriate group-group communication to facilitate plan-sharing, a supergroup can directly employ the primitives. And with respect to primitive-to-group delegation, only the supergroup, with the complete plan, need be the delegation target.

There are two basic circumstances. First, if the member has identity unequal to the group identity, there is no primitive-to-group delegation to be accounted for. Second, if it has equal identity then the identities of all the primitives and groups contained must also be equal, since equality is transitive. A method call from the "outside" is delegated to the supergroup and from there to the primitive objects. Intermediate groups become routers. There is, after all, still only a single *this* and a single *self*.

3. CLASS RELATIONSHIPS

Not all fields and methods of a class belong to instances, and the above classification can apply independently to the static behavior, and their corresponding meaning must be phrased in terms of classes rather than instances:

- We are fortunate that Java provides no way to compare classes for identity[2]. Fortunate, because the fact that Java does not support class equality[3] of differently named classes, which causes great trouble for some Java-generation tools, means that we do not need to eliminate facets, mavericks, and routers as class relationships. But eliminating the "identity" column causes no coalescence of relationships because the "group-to-primitive" column preserves its distinctions.
- The variation of forms for automanipulation refers to the interpretation of *this* in method bodies. Static methods have no *this*, but the analogous meaning for classes applies to how calls on static methods defined by the class itself are handled. The "*this*=primitive" form is interpreted easily as leaving the calls to own class, which are always manifest, as calls to its own class. Likewise, the "*this*=group" form is interpreted by making them refer to the class appropriate to the self= form in use. This can be done by rewriting a copy of the body appropriate to each group from which the static method is called.
- The group-to-primitive forms for "*self*=" must also be reinterpreted without reference to a particular instance. This can be performed, as suggested above, by selecting the appropriate rewriting.

An important case of static behavior is creation. The Java *new* operation (not the constructors that become involved later, during initialization) is equivalent to a static method in the class being instantiated. Creation of an instance of a class may or may not be delegated to a group, which may then call for creation of its parts, including the original.

In general, the class composition form and the instance composition form can be independently selected, so there are actually 7^2=49 kinds of relationships. Of these, perhaps only 13 are of importance, those in which the class composition relation and the instance composition relationships are the same, and those in which the class composition relationship is "stand-alone". We will distinguish these two by prefixing the relationship name with "full-" or "partial-"[4]. When omitted, "partial-" is assumed.

The same constraints on multiple relationships among classes apply as those for instances, and for the same reason. But, for both instances and classes, these constraints can be interpreted either as prohibitions or as reinterpretations of composition operations.

4. COMPOSITION OPERATORS

Groups are created and modified by composition operators. Composition can be described in terms of two operators: ***compose***(relationship,details,group-class-name,object-class-name) and ***compose***(relationship,details,group,object). Both of these operators can produce Java class definitions, and the latter may produce objects and changes to objects as well.

In the discussion of "Objects in Multiple Relationships", certain relationship combinations were noted to be impermissible. That is, however, a static statement. There are two possible ways in which ***compose*** operators could respond to specifying an impermissible combination: the combination could be rejected, or the object could be cloned and the operation performed with respect to its clone. We call these two variant operators: ***compose-two-way*** and ***compose-one-way***.

4.1 Instance Composition and Temporal Instability of Identity – Cloning

The impermissible relationship combinations all arise from incompatible handling of primitive-to-group. And if variant primitive-to-group is forbidden, this is equivalent to the requirement on identity.

So performing a ***compose*** operation for an impermissible combination of instances runs afoul of the conventional idea that identity is an unchanging characteristic[5]. What difficulties can arise from permitting temporal instability in identity? A concrete example occurs if a standalone becomes a facet, router, or maverick of a group, or an object in one of those relationships becomes standalone. Comparisons of its identity with that of the group yield different values after the composition from what they yield before. But facts about the result of this identity test may be presumed externally, and already taken into account in a way that becomes meaningless. This phenomenon is one instance of what we have called "object schizophrenia". A common example of

[2] We are ignoring the library support for reflection. While reflection introduces objects that represent classes, methods, etc., the object is not the class, but only a representative of the class in the current execution. Two different class objects can represent the same class at the same time on two different machines and inequality of class objects is not the same as inequality of classes.

[3] A Java class is either a subclass of, a superclass of, or unrelated to any differently named Java class. Though useful, cycles are not permitted in inheritance graphs.

[4] except in the case of "stand-alone", where they are the same

[5] In fact, there are languages, like Irish, that have two entirely different verb forms for the "changeable is" and the "unchangeable is".

object schizophrenia arises in forming data structures representing sets of objects: no matter how many times an object is added to a set, it is present only once. But what if two objects are added and then they become facets of the same group? The presumed and proven invariant governing the set becomes violated after-the-fact.

However, if it can be assured the group contains at most one facet, maverick or router whose identity it takes, and that no prior remain to other of its facets, mavericks or routers, object-schizophrenia will not arise. Defining ***compose-two-way*** to throw an error after performing the composition is one way of permitting the composition to go ahead but requiring programmers to think about whether they can prove that the identity has not escaped in writing the *catch*. Another, more convenient, solution is to use ***compose-one-way***. The clone it creates is a new object without outstanding uses of its identity.

4.2 Class Composition and Cloning

The same conflicts, with the same potential solutions, arise for class composition as for instance composition, although from different grounds. Classes can always be referenced since their names are available to past and future Java programs with the proper access rights. This means that, except through careful program analysis, developers cannot assure that the exception arising from ***compose-two-way*** can be ignored. Note that this does not mean an object can not be a facet of two groups, only that the two groups must be merged into one larger group so that they are also identical.

5. IMPLEMENTATION NOTES

5.1 Class Composition

Multiple rewriting of a static method has significant cost. The cases in which additional rewritings are needed are noted by shading below in a collapsed version of the table above[6]. If unimplemented, only the 10 relationships: stand-alone, full-associate, associate, aspect[7], affiliate, facet[8], full-maverick, maverick, full-router, and router become available.

Object's relationship to group	Identity	Primitive-to-group	Group-to-primitive	Automanipulation
1. Stand-alone	unequal	no	no	*(this=self=primitive)*
2. Associate	unequal	no	*self*=primitive	*(this=self=primitive)*
3. Aspect	unequal	no	*self*=group	*this*=primitive
4. Affiliate	equal		group	*this*=group
5. Facet	equal	identical	*self*=group	*this*=group
6. Maverick	equal	identical	*self*=primitive	*(this=self=primitive)*
7. Router	equal	identical	no	*(not used)*

[6] "Aspect" with *self*=group, *this*=primitive is implementable without additional rewritings, unless explicit uses of "selfClass" occur in the body. But then, what's the point; it is the same as associate.

[7] AspectJ's "aspect" [4]

[8] full-facet is implemented by Hyper/J [5]

5.2 Instance Composition

Discussion of instance composition presumed that it is possible to treat the call of a method (whether in a group or in a primitive object) from a primitive instance from its call from a group. Discussion of instance composition also presumed that method calls to the group can be distinguished from calls to the primitive objects that are members. The simplest ways of making these distinctions are use two objects, to use two methods, or both. With two methods on two objects, all of the relationships presented can be supported, but without them, some choices are lost.

5.2.1 Instance Composition with a Single Method on Two Objects

The only way of distinguishing calls to an object from a group from calls to the object from outside primitives without coining an additional method is to prevent calls from the outside, managing to substitute the group's identity except in calls from the group. Only in the case of a stand-alone object can the primitive's identity be used outside, and that is because it is never invoked as a group member at all. This voids the columns dealing with identity and primitive-to-group forms, eliminates routers, and renders affiliates and mavericks redundant.

Object's relationship to group	Group-to-primitive	Automanipulation
1. Stand-alone	no	*(this=self=primitive)*
2. Associate ≡ 6. Maverick	*self*=primitive	*(this=self=primitive)*
3. Aspect	*self*=group	*this*=primitive
4. Affiliate		*this*=group
5. Facet ≡ 4. Affiliate		*this*=group
6. Maverick	*self*=primitive	*(this=self=primitive)*
7. Router	no	*(not used)*

5.2.2 Instance Composition with Two Methods on a Single Object

One way of distinguishing calls to objects from a group from calls to the object from outside the group is to use two sets of methods. Using a single object for both the group and its primitives rules out cases in which the identity test should yield "unequal", except in the case of stand-alone objects, which are not part of a group in any case. Despite the fact that coalescing the group object with the member cannot always be employed, it can be used to reduce overheads for facets, mavericks and routers.

Object's relationship to group	Identity	Primitive-to-group	Group-to-primitive	Automanipulation
1. Stand-alone	unequal	no	no	*(this=self=primitive)*
2. Associate	unequal	no	*self*= primitive	*(this=self=primitive)*
3. Aspect	unequal	no	*self*=group	*this*= primitive
4. Affiliate				*this*=group
5. Facet	equal	identical	*self*= group	*this*=group
6. Maverick	equal	identical	*self*= primitive	*(this=self=primitive)*
7. Router	equal	identical	no	*(not used)*

6. RELATED WORK

6.1 Composition Filters

The concept of wrappers and, in particular, wrappers for objects, has long application in software development. Composition filters [1] extend the object-wrapper concept to a group-wrapper. The group embodies dispatch strategies based on its *state* – a set of predicates about the objects in the group. In the classification given above, composition filters are groups. The filtered objects are aspects or full aspects. With composition filters, group behavior is obtained only by directing messages to the group. It has the *compose-two-way* variant of the instance composition operator.

6.2 Subject-Oriented Programming

Subject-Oriented programming [3] introduced the notion that objects in a group can have the same identity and that creation of an instance of one of the member classes causes creation of the group. The member is a class facet, although the creation need not be delegated to all members. As discussed above, this implies that the group should handle the messages directed to its members. In SOP, the subjects are all full facets.

6.3 Objects in Groups

Doug Lea has written a survey on objects in groups [7] recapping prior work. He also presents an alternative delivery model relying on *channels* rather than on object identity to describe the target for the message. The introduction of channels does not change the basic form for the analysis presented above, but does allow for many more mixed or intermediate cases in the analysis.

6.4 Aspect-Oriented Programming

Aspect-Oriented Programming [6] retained the concept that creation of an instance of one of the member classes causes creation of the group. But it does so for only one of the member classes, called the base. Other member classes are treated like members of composition filters. In AOP, the base and the aspects have different relationships to the group. The base is a full facet but the aspects are full aspects. It has the *compose-two-way* class composition operator. AspectJ [5] provides a realization of AOP in which the group and the facet are coalesced into a single object.

6.5 Hyper/J

Hyper/J [9] is a realization of MDSOC [13], an evolution from SOP. It has both the *compose-two-way* and the *compose-one-way* variants of the class composition operator. In Hyper/J, the group and all the facets are coalesced into a single object

6.6 Compound References

Ostermann and Mezini [10] identified a number of separate composition properties, subsets of which are usually bundled together to form composition mechanisms like inheritance and delegation. They showed that use of more powerfully interpreted references, called *compound references*, allows flexible combination of these properties and provides important semantic options not traditionally available. While shifting discussion of dispatch from objects to generalized references provides an important alternative to group formation, this paper deals with solutions within the more traditional view of object identity and reference.

7. REFERENCES

[1] Aksit M., Bergmans L., Vural S. An object-oriented language-database integration model: the composition-filters approach. In *Proceedings of the European Conference on Object-Oriented Programming*. Springer Verlag, 1992

[2] Gamma, E., Helm, R., Johnson, R., Vlissides, J. *Design Patterns*. Addison-Wesley, 1995

[3] Harrison, W., and Ossher, H. Subject-Oriented Programming - A Critique of Pure Objects. In *Proceedings of the 1993 Conference on Object-Oriented Programming Systems, Languages, and Applications*. September 1993

[4] Helm, R., Holland, I., and Gangopadhyay, D., "Contracts: Specifying Behavioral Compositions in Object-Oriented Systems", Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications, (Vancouver), ACM, October 1990

[5] Kiczales, G. Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., and Griswold, W.G. An overview of AspectJ. In *Proceedings of the European Conference on Object-Oriented Programming*. Finland, 1997

[6] Kiczales, G. Lamping, J., Mendhekar, A., Maeda, C., Videira Lopes, C., Loingtier, J-M. Irwin, J. Aspect-oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming*. Finland, 1997. Invited presentation.

[7] Lea D., *Objects in Groups*, December, 1993, http://gee.cs.oswego.edu/dl/groups/groups.html

[8] Ossher, H. and Harrison, W., "Support for Change in RPDE^3", Proceedings of the Fourth ACM SIGSOFT Symposium on Software Development Environments, pp. 218-228, Irvine CA, December 1990

[9] Ossher, H. and Tarr, P. "Multi-dimensional separation of concerns and the Hyperspace approach." In *Software Architectures and Component Technology* (M. Aksit, ed.), 293–323, Kluwer, 2002.

[10] Ostermann, K., Mezini, M. Object-Oriented Composition Untangled. In *Proceedings of the 2001 Conference on Object-Oriented Programming Systems, Languages, and Applications*. October 2001.

[11] Reiss, S., "Connecting Tools Using Message Passing in the Field Environment", IEEE Software, pp. 57-66, July 1990.

[12] Sullivan, K.J. and Notkin, D., ``Reconciling Environment Integration and Software Evolution," *ACM Transactions on Software Engineering and Methodology* 1(3), pp. 229-268, July, 1992.

[13] Tarr, P., Ossher, H., Harrison, W. and Sutton, Jr., S. M., "*N* degrees of separation: Multi-dimensional separation of concerns." In *Proceedings of the 21st International Conference on Software Engineering (ICSE '99)*, 107–119, IEEE, May 1999.

[14] Web site, "Subject-Oriented Programming and Design Patterns," http://www.research.ibm.com/sop/sopcpats.htm

Compilation Semantics of Aspect-Oriented Programs

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

Gregor Kiczales
Department of Computer Science
University of British Columbia
gregor@cs.ubc.ca

Chris Dutchyn
Department of Computer Science
University of British Columbia
cdutchyn@cs.ubc.ca

ABSTRACT

This paper presents a semantics-based compilation framework for an aspect-oriented programming language based on its operational semantics model. Using partial evaluation, the framework can explain several issues in compilation processes, including how to find places in program text to insert aspect code and how to remove unnecessary run-time checks. It also illustrates optimization of calling-context sensitive pointcuts (`cflow`), implemented in real compilers.

Keywords

Aspect SandBox, dynamic join point model, partial evaluation, Futamura projection, compile-time weaving, context-sensitive pointcut designators (`cflow`)

1. INTRODUCTION

This work is part of a larger project, the Aspect SandBox (ASB), that aims to provide concise models of aspect-oriented programming (AOP) for theoretical studies and to provide a tool for prototyping alternative AOP semantics and implementation techniques. To avoid difficulties to develop formal semantics directly from artifacts as complex as AspectJ and Hyper/J, ASB provides a suite of interpreters of simplified languages. Those languages have sufficient features to characterize existing AOP languages. In this paper we report one result from the ASB project—a semantics-based explanation of the compilation strategy for advice dispatch in AspectJ like languages[6, 7, 11, 12].

The idea is to use partial evaluation to perform as many tests as possible at compile-time, and to insert applicable advice bodies directly into the program. Our semantic framework also explains the optimization used by the AspectJ compiler for context-sensitive pointcuts (`cflow` and `cflowbelow`).

Some of the issues our semantic framework clarifies include:

- The mapping between dynamic join points and the points in the program text, or *join point shadows*, where the compiler actually operates.
- What dispatch can be 'compiled-out' and what must be done at runtime.
- The performance impact different kinds of advice and pointcuts can have on a program.
- How the compiler must handle recursive application of advice.

1.1 Join Point Models

Aspect-oriented programming (AOP) is a paradigm to modularize crosscutting concerns[13]. An AOP program is effectively written in multiple modularities—concerns that are local in one are diffuse in another and vice-versa. Thus far, several AOP languages are proposed from practical to experimental levels[3, 11, 12, 16, 17].

The ability of an AOP language to support crosscutting lies in its *join point model* (JPM). A JPM consists of three elements:

- The *join points* are the points of reference that aspect programs can use to refer to the computation of the whole program. *Lexical* join points are locations in the program text (*e.g.,* "the body of a method"). *Dynamic* join points are run-time entities, such as events that take place during execution of the program (*e.g.,* "an invocation of a method").
- A *means of identifying* join points. (*e.g.,* "the bodies of methods in a particular class," or "all invocations of a particular method")
- A *means of specifying semantics* at join points. (*e.g.,* "run this code beforehand")

As an example, in AspectJ:

*This work is carried out during his visit to University of British Columbia.

- the join points are nodes in the runtime control flow graph of the program, such as when a method is called (and returns), and when a field is read (and the value is returned). (*e.g.,* "a call to method `set(int)` of class `Point`"[1])
- the means of identifying join points is the *pointcut* mechanism, which can pick out join points based on things like the name of the method, the package, the caller, and so forth. (*e.g.,* "`call(void Point.set(int))`")
- the means of specifying semantics is the *advice* mechanism, which makes it possible to specify additional code that should run at join points.
  (*e.g.,* "`before : call(void Point.set(int)) { Log.add("set"); }`")

In this paper, we will be working with a simplified JPM similar to the one from AspectJ. (See Section 2.1 for details.)

The rest of the paper is organized as follows. Section 2 introduces our AOP language, AJD, and shows its interpreter. Section 3 presents a compilation framework for AJD excluding context-sensitive pointcuts, which are deferred to Section 4. Section 5 relates our study to other formal studies in AOP and other compilation frameworks. Section 6 concludes the paper with future directions.

2. AJD: DYNAMIC JOIN POINT MODEL AOP LANGUAGE

This section introduces our small AOP language, AJD, which implements core features of AspectJ's dynamic join point model. The language consists of a simple object-oriented language and its AOP extension. Its operational semantics is given as an interpreter written in Scheme. A formalization of a procedural subset of AJD is presented by Wand and the second and the third authors[20].

2.1 Informal Semantics

We first informally present the semantics of AJD. In short, AJD is an AOP language based on a simple object-oriented language with classes, objects, instance variables, and methods. Its AOP features covers essential part of AspectJ (version 1.0).

2.1.1 Object Semantics

Figure 1 is an example program. For readability, we use a Java-like syntax in the paper[2]. It defines a `Point` class with one integer instance variable `x`, a unary constructor, and three methods `set`, `move` and `main`.

When method `main` of a `Point` object is executed, line 7 creates another `Point` object and runs the constructor defined at line 3. Line 8 invokes method `move` on the created object. Finally, line 9 reads and displays the value of variable `x` of the object.

[1]For simplicity later in the paper, we are using one-dimensional points as an example.
[2]Our implementation actually uses an S-expression based syntax.

```
1  class Point {
2    int x;
3    Point(int ix)    { this.set(ix); }
4    void set(int newx) { this.x = newx; }
5    void move(int dx) { this.set(this.x + dx); }
6    void main() {
7      Point p = new Point(1);
8      p.move(5);
9      write(p.x); newline();
10   }
11 }
```

Figure 1: An Example Program. (`write` and `newline` are primitive operators.)

$p \in \{\text{pointcuts}\}$, $m \in \{\text{method signatures}\}$,
$n \in \{\text{constructor signatures}\}$, $v \in \{\text{identifiers with types}\}$

$$
\begin{array}{rcl}
p & ::= & \texttt{call}(m) \mid \texttt{execution}(m) \mid \texttt{new}(n) \\
& \mid & \texttt{target}(v) \mid \texttt{args}(v, \ldots) \mid p\texttt{\&\&}p \mid p\texttt{||}p \mid \texttt{!}p \\
& \mid & \texttt{cflow}(p) \mid \texttt{cflowbelow}(p)
\end{array}
$$

Figure 2: Syntax of Pointcuts.

2.1.2 Aspect Semantics

To explain the semantics of AOP features in AJD, we first define its JPM.

2.1.2.1 Join Point

The *join point* is an action during program execution, including method calls, method execution, object creation, and advice execution. (Note that a method invocation is treated as a call join point at the caller's side and an execution join point at the receiver's side.) The *kind* of the join point is the kind of action (*e.g.,* call and execution).

2.1.2.2 Means of Identifying Join Points

The means of identifying join points is the pointcut mechanism. A *pointcut* is a predicate on join points, which is used to specify the join points that a piece of advice applies to. The syntax of pointcuts is shown in Figure 2. Since pointcuts can have parameters, the evaluation of a pointcut with respect to a join point results in either bindings that satisfy the pointcut (meaning true), or false.

The first three pointcuts (`call`, `execution`, and `new`) match join points that have the same kind and signature as the pointcut. The next two pointcuts (`target` and `args`) match any join point that has values of specified types. The next three operators (`&&`, `||` and `!`) logically combine or negate pointcuts. The last two pointcuts match join points that have a join point matching their sub-pointcuts in the call-stack. These are discussed in Section 4 in more detail. Interpretation of pointcuts is formally presented in other literature[20].

2.1.2.3 Means of Specifying Semantics

The means of specifying semantics is the advice mechanism. A piece of *advice* contains a pointcut and a body expression. When a join point is created, and it matches the pointcut of a piece of advice, the body of the advice is executed. There

```
1 before : call(void Point.set(int)) && args(int z) {
2   write("set:"); write(z); newline();
3 }
```

Figure 3: Example Advice.

```
1  (define eval
2    (lambda (exp env jp)
3      (cond
4       ((const-exp? exp) (const-value exp))
5       ((var-exp? exp) (lookup env (var-name exp)))
6       ((call-exp? exp)
7        (call (call-signature exp)
8              (eval (call-target exp) env jp)
9              (eval-rands (call-rands exp) env jp)
10             jp))
11      ...)))
12
13 (define call
14   (lambda (sig obj args jp)
15     (execute (lookup-method (object-class obj) sig)
16              obj args jp)))
17
18 (define execute
19   (lambda (method this args jp)
20     (let ((class (method-class method))
21           (params (method-params method)))
22       (eval (method-body method)
23             (new-env (list* 'this '%host params)
24                      (list* this class args))
25             jp))))
```

Figure 4: Expression Interpreter.

are two types of advice, namely `before` and `after`. A `before` advice is executed before the original action is taken place. Similarly, the `after` is executed after the original action is completed.

Figure 3 shows an example of advice that lets the example program to print a message before every call to method `set`. The keyword `before` specifies the type of the advice. pointcut is written after the colon. The pointcut matches join points that call method `set` of class `Point`, and the `args` sub-pointcut binds variable `z` to the argument to method `set`. Line 2 is the body, which prints messages and the value of the argument.

When the program in Figure 1 is executed together with the advice in Figure 3, the advice matches to the call to `set` twice (in the constructor and in method `set`), it thus will print "`set:1`", "`set:6`" and "`6`".

2.2 AJD Interpreter

The interpreter of AJD consists of an expression interpreter and several definitions for AOP features including the data structure for a join point, wrappers for creating join points, a weaver, and a pointcut interpreter.

2.2.1 Expression Interpreter

Figure 4 shows the core of the expression interpreter excluding support for AOP features. The main function `eval` takes an expression, an environment, and a join point as its parameters. The join point is an execution join point at the enclosing method or constructor.

An expression is a parsed abstract syntax tree. There are predicates (*e.g.,* `const-exp?` and `call-exp?`) and selectors (*e.g.,* `const-value` and `call-signature`) for the syntax trees. An environment binds variables to mutable cells; *i.e.,* an assignment to a variable is implemented as side-effect in Scheme. An object is a Scheme data structure that has a class information and mutable fields for instance variables. Likewise, an assignment to an instance variable is implemented as side-effect.

Each action that creates join points is defined as a separate sub-function, so that we can add AOP support later.

For example, the interpreter evaluates a method call expression in the following manner. First, sub-expressions for the target object and operand values are recursively evaluated (ll.8–9). Next, in function `call`, a method is looked-up in the class of the target object (l.15). Then, in function `execute`, an environment that binds the formal parameters to the operand values is created (ll.23–24)[3]. Finally, the body of the method is evaluated with newly created environment (ll.22–25).

field	available information
`kind`	`call`, `execution`, etc.
`name`	name of method or class
`target`	target of method invocation
`args`	arguments to a method
`stack`	(explained in Section 4)

Table 1: Fields in Join Points

2.2.2 Join Point

A join point is a data structure that is created upon an action in the expression interpreter. A piece of advice obtains all information about advised action from join points. In our implementation, a join point is a record of `kind`, `name`, `target`, `args`, and `stack`. Table 1 summarizes values in those fields. There are selectors, such as `jp-kind`, and a constructor, `make-jp`, for accessing and creating join points.

2.2.3 Wrapper

In order to advice actions performed in the expression interpreter, we wrap the interpreter functions so that they (conceptually) create dynamic join points. Figure 5 shows how `call`—one of such a function—is wrapped. When a method is to be called, the function first creates a join point

```
1 (define call
2   (lambda (sig obj args jp)
3     (weave (make-jp 'call sig obj args jp)
4            (lambda (args jp)
5              ;; body of the original call goes here
6              )
7            args)))
```

Figure 5: A Wrapper.

[3]The pseudo-variable `%host` is used for looking-up methods for super classes.

```
1  (define weave
2    (lambda (jp action args)
3      (call-befores/afters *befores* args jp)
4      (let ((result (action args jp)))
5        (call-befores/afters *afters* args jp)
6        result)))
7
8  (define call-befores/afters
9    (lambda (advs args jp)
10     (for-each (call-before/after args jp) advs)))
11
12 (define call-before/after
13   (lambda (args jp)
14     (lambda (adv)
15       (let ((env (pointcut-match? (advice-pointcut adv)
16                                   jp)))
17         (if env
18             (execute-before/after adv env jp))))))
19
20 (define execute-before/after
21   (lambda (adv env jp)
22     (weave (make-jp 'aexecution adv #f #f '() jp)
23            (lambda (args jp)
24              (eval (advice-body adv) env jp))
25            '())))
```

Figure 6: Weaver.

that represents the call action (l.3) and applies it to `weave`, which executes advice applicable to the join point (explained below). The lambda-closure passed to `weave` (ll.4–6) defines the action of `call`, which is executed during the weaving process.

Likewise, the other functions including method execution, object creation, and advice execution (defined later) are wrapped.

2.2.4 Weaver

Figure 6 shows the definition of the weaver. Function `weave` takes a join point (`jp`), a lambda-closure for continuing the original action (`action`), and a list of arguments to `action` (`args`). It also uses advice definitions in global variables (`*befores*` and `*afters*`). It defines the order of advice execution; it executes `before`s first, then the original action, followed by `after`s last.

Function `call-befores/afters` processes a list of advice. It matches the pointcut of each piece of advice against the current join point (ll.15–16), and executes the body of the advice if they match (ll.17–18). In order to (potentially) advise execution of advice, the function `execute-before/after` is also wrapped. Line 24 actually executes the advice body in an environment that provides the bindings expressed by the pointcut.

Calling `around` advice has basically the same structure for the `before` and `after`. It is, however, more complicated due to its interleaved execution for the `proceed` mechanism.

2.2.5 Pointcut interpreter

```
1  (define pointcut-match?
2    (lambda (pc jp)
3      (cond
4       ((and (call-pointcut? pc) (call-jp? jp)
5             (sig-match? (pointcut-sig pc) (jp-name jp)))
6        (make-env '() '()))
7       ((and (args-pointcut? pc)
8             (types-match? (jp-args jp)
9                           (pointcut-arg-types pc)))
10       (make-env (pointcut-arg-names pc) (jp-args jp)))
11      ...
12      (else #f))))
```

Figure 7: Pointcut Interpreter.

The pointcut interpreter `pointcut-match?`, shown in Figure 7, matches a pointcut to a join point. Due to space limitations, we only show rules for two types of pointcuts. The first rule (ll.4–6) defines that a `call`(m) pointcut matches to a `call` join point that whose `name` field matches to m. It returns an empty environment that represent 'true' (l.6). An `args`($t\ x, \ldots$) pointcut (where t and x are a type and a variable, respectively) matches to any join point whose arguments have the same type to $t, \ldots$ (ll.7–10). It returns an environment that binds variable $x, \ldots$ in the pointcut to the value of the argument in the join point (l.10). False is returned when matching fails (l.12).

3. COMPILING AJD PROGRAMS BY PARTIAL EVALUATION

3.1 Outline

Our compilation framework is based on partial evaluation of an interpreter, which is known as *the first Futamura projection*[9]. Given an interpreter of a language and a program to be interpreted, partially evaluating the interpreter with respect to the subject program generates a compiled program (called a *residual* program). Following this scheme, we can expect that partial evaluation of an AOP interpreter with respect to a subject program *and advice definitions* would generate a compiled, or *statically woven* program.

While the AJD interpreter is written as to 'test-and-execute' *all* pieces of advice at each *dynamic* join point, our compilation framework successfully inserts *only applicable* advice to each shadow of join points. This is achieved in the following way:

1. Our compilation framework runs a partial evaluator with AJD interpreter and each method definition.
2. The partial evaluator processes the expression interpreter, which virtually walks over the expressions in the method. All shadows of join points are thus instantiated.
3. At each shadow of join points, the partial evaluator further processes the weaver. Using statically given advice definitions, it (conceptually) inserts test-and-execute sequence of all advice.
4. For each piece of advice, the partial evaluator reduces the test-and-execute code into a conditional branch

that has either constant or dynamic value as its condition, and the advice body as its then-clause. Depending on the condition, the entire code or the test code may be removed.

5. The partial evaluator processes the code that executes the advice body. It thus instantiates shadows of join points in the advice body. The steps from 3 recursively compiles 'advised advice execution.'

As is mentioned in the above step 1, we run the partial evaluator with respect to each method definition. This is because the applicable method for a method call can not be determined at compile-time in object-oriented languages. Therefore, we start the partially evaluator with the `execute` function and its `method` parameter. The rest of the parameters (`env` and `jp`) are set to unknown at partial evaluation time. The residual program serves as a compiled (or statically woven) code of the method written in Scheme. The function is stored in a dispatch table so that it will be directly called at run-time.

For partial evaluation, we used PGG, an offline partial evaluator for Scheme[19].

3.2 How AJD is Partially Evaluated

An offline partial evaluator processes a program in the following way. It first annotates expressions in the program as either *static* or *dynamic*, based on their dependency on the statically known parameters. Those annotations are often called *binding-times*. It then processes the program from the beginning by actually evaluating static expressions and by returning symbolic expressions for dynamic expressions. The resulted program, which is called *residual program*, consists of dynamic expressions in which statically computed values are embedded.

This subsection explains how the AJD interpreter is partially evaluated with respect to a subject program, by emphasizing what operations can be performed at partial evaluation time. Although the partial evaluation is an automatic process, we believe understanding this process is crucially important for identifying compile-time information and also for developing better insights into design of hand-written compilers.

3.2.1 Compilation of Expressions

The essence of the Futamura projection is to perform computation involving `exp` at partial evaluation time. Specialization of `execute` with each static `method` makes `eval` of `exp` static, and subsequent execution keeps this static property of `exp`. In contrast, `call` applies the `method` parameter as a dynamic value to `execute` due to the nature of dynamic dispatching in object-oriented languages. We therefore configure[4] the partial evaluator not to process `execute` from `call` so that it will not 'downgrade' the binding-time of `exp` to dynamic.

The environment (`env`) is treated as dynamic. With more careful interpreter design, we could make it *partially-static* data, in which variable names are static and values are dynamic. However, this is not the focus of this paper.

[4]To do so, we rewrite `call` to call `execute*` instead of `execute`, and manually give dynamic binding-time to `execute*`.

3.2.2 Compilation of Advice

As is mentioned in Section 3.1, our compilation framework inserts advice bodies into their applicable shadows of join points with appropriate guards. Below, we explain how this is done by the partial evaluator.

1. A wrapper (*e.g.,* Figure 5) creates a join point upon calling `weave`. The first two fields of the join point, namely `kind` and `name`, are static because they merely depend on the program text. The rest fields have values computed at run-time. Those static fields could be passed to the weaver either by using partially-static data structure[4] or by rewriting the program to keep those three values in a split data structure. We took the latter approach for the ease of debugging and also for other technical reasons.

2. Function `weave` (Figure 6) is executed with a partially static join point, an action, and dynamic arguments. Since the advice definitions are statically available, the partial evaluator unrolls loops that test each advice definition (*i.e.,* `for-each` in `eval-befores/afters`).

3. As explained in Section 3.2.3, matching a static pointcut to a partially static join point may result in either a static or dynamic value. Therefore, the test-and-execute sequence (in `eval-before/after`) becomes one of the following three:

   Statically false: No action is taken; i.e., no code is inserted into compiled code.

   Statically true: The body of the advice is partially evaluated; *i.e.,* the body is inserted in compiled code without guards.

   Dynamic: In this case, partial evaluation of `pointcut-match?` generates an `if` expression whose then-clause is the above 'statically true' case and the else-clause is 'statically false' case. Essentially, the advice body is inserted with a guard.

4. In the statically true or dynamic cases at the above step, the partial evaluator processes the evaluation of the advice body. Since the wrapper of the advice-execution calls `weave`, application of advice to the advice body is also compiled.

5. When the original action is evaluated (l.4 in Figure 6), the residual code of the original action is inserted. This residual code from `weave` will thus have the original computation surrounded by applicable advice bodies.

3.2.3 Compilation of Pointcut

In step 3 above, pointcut interpreter (Figure 7) is partially evaluated with a static pointcut and static fields in a join point. The partial evaluation process depends on the type of the pointcut. For pointcuts that depend on only static fields of a join point (namely `call`, `execution` and `new`), the condition is statically computed to either an environment (as true) or false. For pointcuts that test values in the join point

```
1  (define point-move
2    (lambda (this1 args2 jp3)
3      (let* ((jp4 (make-jp 'execution 'move
4                           this1 args2 jp3))
5             (args5 (list (+ (get-field this1 'x)
6                             (car args2))))
7             (jp6 (make-jp 'call 'set
8                           this1 args5 jp4)))
9        (if (type-match? args5 '(int))
10           (begin (write "set:")
11                  (write (car args5)) (newline)))
12       (execute* (lookup-method (object-class this1)
13                                'set)
14                 this1 args5 jp6))))
```

Figure 8: Compiled code of `move` method of `Point` class.

(namely `target` and `args`), the partial evaluator returns residual code that dynamically tests the types of the values in the join point. For example, when `pointcut-match?` is partially evaluated with respect to `args(int x)`, the following expression is returned as residual code.

```
1  (if (types-match? (jp-args jp) '(int))
2      (make-env '(x) (jp-args jp))
3      #f)
```

Logical operators (namely `&&`, `||` and `!`) are partially evaluated into an expression that combines the residual expressions of its sub-pointcuts. The remaining two pointcuts (`cflow` and `cflowbelow`) are discussed in the next section.

The actual `pointcut-match?` is written in a continuation-passing style so that partially evaluator can reduce a conditional branch in the weaver (ll.17–18 in Figure 6) for the static cases. This is a standard technique in partial evaluation.

3.3 Compiled Code

Figure 8 shows the compiled code for the `move` method defined in Figure 1 combined with the advice given in Figure 3. For readability, we post-processed the residual code by eliminating dead code, propagating constants, renaming variable names, resolving environment accesses, and so forth. The join points combine both static and dynamic fields for readability, while they are manually split in the actual implementation.

It first creates a join point for the method execution (ll.3–4), a parameter list (ll.5–6) and a join point (ll.7–8) for the method call. Lines 9 to 11 are advice body with a guard. The guard checks the residual condition for `args` pointcut. (Note that no run-time checks are performed for `call` pointcut.) If matched, the body of the advice is executed(ll.10–11). Finally, the original action (*i.e.,* method call) is performed (ll.12–14).

As we see, advice execution is successfully compiled. Even though there is a shadow of `execution` join points at the beginning of the method, no advice bodies are inserted in the compiled function as it does not match any advice.

```
1  after : call(void Point.set(int))
2          && cflow(call(void Point.move(int))
3                   && args(int w)) {
4    write("under move:"); write(w); newline();
5  }
```

Figure 9: Advice with `cflow` Pointcut.

4. COMPILING CALLING-CONTEXT SENSITIVE POINTCUTS

As briefly mentioned before, `cflow` and `cflowbelow` pointcuts can investigate join points in the call-stack; *i.e.,* their truth value is sensitive to calling context. Here, we first show a straightforward implementation that is based on a stack of join points. It is inefficient, however, and can not be compiled properly.

We then show a more optimized implementation that can be found in AspectJ compiler. The implementation exploits incremental natures of those pointcuts, and shown as a modified version of AJD. We can also see those pointcuts can be properly compiled by using our compilation framework.

To keep discussion simple, we only explain `cflow` in this section. Extending our idea to `cflowbelow` is easy and actually done in our experimental system.

4.1 Calling-Context Sensitive Pointcut: cflow

A pointcut `cflow`(p) matches to any join points if there is a join point that matches to p in its call-stack. Figure 9 is an example. The `cflow` pointcut in lines 2–3 specifies join points that are created during the method call to `move`. When this pointcut matches a join point, the `args(int w)` sub-pointcut gets the parameter to `move` from the stack.

As a result, execution of the program in Figure 1 with pieces of advice in Figures 3 and 9 prints "`set:1`" first, "`set:6`" next, and then "`under move:5`" followed by "`6`" last. The call to `set` from the constructor is not advised by the advice using `cflow`.

4.2 Stack-Based Implementation

A straightforward implementation is to keep a stack of join points and to examine each join point in the stack from the top when `cflow` is evaluated.

We use the `stack` field in a join point to maintain the stack. Whenever a new join point is created, we record previous join point in the `stack` field (as is done as the last argument to `make-jp` in Figure 5). Since join points are passed along method calls, the join points chained by the `stack` field from the current one form a stack of join points. Restoring old join points is implicitly achieved by merely using the original join point in the caller's continuation.

The algorithm to evaluate `cflow`(p) simply runs down the stack until it finds a join point that matches to p, as shown in Figure 10. If it reaches the bottom of the stack, the result is false.

The problem with this implementation is run-time overhead.

```
1  (define pointcut-match?
2    (lambda (pc jp)
3      (cond ...
4          ((and (cflow-pointcut? pc)
5                (not (bottom? jp)))
6           (or (pointcut-match? (pointcut-body pc) jp)
7               (pointcut-match? pc (jp-stack jp))))
8          ...)))
```

Figure 10: Naive Algorithm for Evaluating `cflow`.

In order to manage the stack, we have to push[5] a join point each time a new join point is created. Evaluation of `cflow` takes linear time in the stack depth at worse. When `cflow` pointcuts in a program match only specific join points, keeping the other join points in the stack and testing them is waste of time and space.

Our compilation does not help those problems. Rather, it highlights the problems. Since relationship between caller and receiver is unknown to the partial evaluator, the `stack` field of a join point becomes dynamic. Consequently, a stack of join points becomes partially-static in which only some fields of the topmost element are static, while the other elements are totally dynamic. When partial evaluator processes `pointcut-match?` with a static `cflow` pointcut and a partially static join point, the second recursive call (l.7 in Figure 10) supplies a dynamic (not partially static) join point. This makes the residual code a loop that dynamically tests each join point in the stack except for the top element[6]; *i.e.,* all the tests involving with `cflow` are performed at runtime.

4.3 State-Based Implementation

A more optimized implementation of `cflow` in AspectJ compiler is to exploit its incremental nature. This idea can be explained by an example. Assume (as in Figure 9) that there is pointcut "`cflow(call(void Point.move(int)))`" in a program. The pointcut becomes true once `move` is called. Then, until the control returns from `move` (or another call to `move` is taken place), the truth value of the pointcut is unchanged. This means that the system needs only manage the state of each `cflow`(p) and update that state at the beginning and the end of join points that make p true. Note that the state should be managed by a stack because it may be rewound to its previous state upon returning from actions.

This state-based optimization can be explained in the following regards:

- The state-based implementation avoids repeatedly matching `cflow` bodies to the same join point in the stack, which can happen in the stack-based implementation. This is achieved by evaluating bodies of `cflow` at each join point in advance, and records the result as its state for later use.

[5]By having a pointer to 'current' join point in parameters to each function, pop can be automatically done by returning from the function.

[6]If the partial evaluator supported polyvariant specialization[5]. Otherwise, the test for the topmost element is also coerced to dynamic.

```
1  (define weave
2    (lambda (jp action args)
3      (let ((new-jp (update-states *cflow-pointcuts*
4                                   jp)))
5        ...the body of original weave...
6        )))
7
8  ;;; fold: (α × β → α) × α × β list → α
9  (define update-states
10   (lambda (pcs jp)
11     (fold (lambda (pc njp)
12             (let ((env (pointcut-match?
13                         (pointcut-body pc jp))))
14               (if env
15                   (update-state
16                    njp (pointcut-id pc) env)
17                   njp)))
18           jp pcs)))
19
20 (define pointcut-match?
21   (lambda (pc jp)
22     (cond ...
23         ((cflow-pointcut? pc)
24          (lookup-state jp (pointcut-id pc)))
25         ...)))
```

Figure 11: State-based Implementation of `cflow`. (`update-state` *jp id new-state*) returns a copy of *jp* in which *id*'s state is changed to *new-state*. (`lookup-state` *jp id*) returns the state of *id* in *jp*.

- The state-based implementation makes static evaluation (*i.e.,* compilation) of `cflow` bodies possible, which can not in the stack-based implementation. This is because bodies are evaluated at each shadow of join points.
- The state-based implementation usually performs a smaller number of stack operations because the state of a `cflow` pointcut needs not be updated at the join points not matching to its body. On the other hand, the stack-based implementation has to push all join points on the stack.
- The state-based implementation evaluate `cflow` pointcut in constant time in by having a stack of states for each `cflow` pointcut.

It is not difficult to implement this idea in AJD. Figure 11 outlines the algorithm. Before running a subject program, the system collects all `cflow` pointcuts in the program, including those appear inside of other `cflow` pointcuts. The collected pointcuts are stored in a global variable `*cflow-pointcuts*`. The system also gives unique identifiers to them, which are accessible via `pointcut-id`. We rename the last field of a join point from `stack` to `state`, so that it stores the current states of all `cflow` pointcuts.

When the interpreter creates a join point, it also updates the states of all `cflow` pointcuts by wrapping `weave`. The wrapper creates a copy of the new join point with updated `cflow` states (ll.3–4), and performs the original action. Function `update-states` evaluates the sub-pointcut of each `cflow`

```
1  (let* ((val7 ...create a point object ...)
2         (args9 '(5))
3         (jp8 (make-jp this1 args9 (jp-state jp3))))
4    (if (types-match? args9 '(int))
5        (begin
6          (execute* (lookup-method (object-class val7)
7                                   'move)
8                    val7 args9
9                    (state-update jp8 '_g1
10                                 (new-env '(w) args9)))
11         ... write and newline ...)
12       ... omitted ...))
```

Figure 12: Compiled code of "`p.move(5)`" with `cflow` advice. (Definitions of variables `env6`, `this1` and `jp` are omitted.)

```
1  (define point-move
2   (lambda (this1 args2 jp3)
3    (let* ((args5 (list (+ (get-field this1 'x)
4                           (car args2))))
5           (jp6 (make-jp this1 args5 (jp-state jp3))))
6     (if (types-match? args5 '(int))
7       (begin
8         (write "set:") (write (car args5)) (newline)
9         (let* ((val7
10                (execute* (lookup-method
11                           (object-class this1) 'set)
12                          this1 args5 jp6))
13               (env8 (state-lookup jp6 '_g1)))
14          (if env8
15            (begin (write "under move:")
16                   (write (lookup env8 'w)) (newline)))
17          val7))
18       ...omitted...))))
```

Figure 13: Compiled code of method `move` with `cflow` advice.

pointcut (ll.12–13), and updates the state if the result is true (ll.15–16).

Interpretation of `cflow` pointcut is merely looks up the state in the current join point (l.24).

Implementation of `cflowbelow` pointcuts is straightforward if we notice that a current value of `cflowbelow`(p) is the value of `cflow`(p) for the *previous* join point. The implementation has a pair of states for each `cflowbelow`(p) pointcut: the one is the state of `cflow`(p) regarding to the current join point and the other is the state of `cflow`(p) regarding to the previous join point, which is the truth value of `cflowbelow`(p) for the current join point.

4.4 Compilation Result

Figures 12 and 13 show excerpts of compiled code for the program in Figure 1 with the two pieces of advice in Figures 3 and 9. The compiler gave `_g1` to the `cflow` pointcut as its identifier.

Figure 12 corresponds to "`p.move(5);`" (l.8 in Figure 1). Since the method call to `move` makes the `cflow` to true, the compiled code updates the state of `_g1` to an environment created by `args` pointcut in the join point (ll.9–10), and passes the updated join point to the method.

Figure 13 shows the compiled `move` method. We can see the additional code for the advice using `cflow` at lines 13–16. It dynamically evaluates the `cflow` pointcut by merely looking its state up, and runs the body of advice if the pointcut is true. The value of variable `w`, which is bound by `args` pointcut in `cflow`, is taken from the recorded state of `cflow` pointcut. Since the state is updated when `move` is to be called, it gives the argument value to `move` method.

To summarize, our framework compiles a program with `cflow` pointcuts into one with state update operations at each join point that matches the sub-pointcut of each `cflow` pointcut, and state look-ups in the guard of advice bodies. In terms of run-time checks for pointcuts, the code is basically identical to the one generated by AspectJ compiler.

5. RELATED WORK

In reflective languages, some crosscutting concerns can be controlled through meta-programming[10, 18]. Several researchers including the first author have successfully compiled reflective programs by using partial evaluation[2, 14, 15]. It is more difficult, however in reflective languages, to ensure successful compilation because the programmer can easily write a meta-program that confuses the partial evaluator.

Wand and the second and the third authors presented a formal model of the procedural version of AJD[20]. Our model is based on this, and used it for compilation and optimizing `cflow` pointcuts.

Douence et al. showed an operational semantics of an AOP system[8]. Their system is based on a 'monitor' that observes the behavior of subject program, and the weaving is triggered by means of pattern matching to a stream of events. They also gave a program transformation system that inserts code to trigger the monitor into subject program. Our framework automatically performs this insertion by using partial evaluation.

Andrews proposed process algebras as a formal basis of AOP languages[1]. In his system, advice execution is represented by synchronized processes, and compilation (static weaving) is transformation of processes that removes synchronization. Our experience suggests that powerful transformation techniques like partial evaluator would be needed to effectively remove run-time checks in dynamic join point models.

6. CONCLUSION AND FUTURE WORK

In this paper, we presented a compilation framework to an aspect-oriented programming (AOP) language, AJD, based on operational semantics and partial evaluation techniques. The framework explains issues in AOP compilers including identifying shadows of join points, compiling-out pointcuts and recursively applying advice. The optimized `cflow` implementation in AspectJ compiler can also be explained in this framework.

The use of partial evaluation allows us to keep simple oper-

ational semantics in which everything is processed at run-time, and to relate the semantics to compilation. Partial evaluation also allows us to understand the data dependency in our interpreter by means of its binding-time analysis. We believe this approach would be also useful to prototyping new AOP features with effective compilation in mind.

Although our language supports only core features of practical AOP languages, we believe that this work could bridge between formal studies and practical design and implementation of AOP languages.

Future directions of this study could include the following topics. Optimization algorithms could be studied for AOP programs based on our model, for example, elimination of more run-time checks with the aid of static analysis. Our model could be refined into more formal systems so that we could relate between semantics and compilation with correctness proofs. Our system could also be applied to design and test new AOP features.

7. REFERENCES

[1] James H. Andrews. Process-algebraic foundations of aspect-oriented programming. In Yonezawa and Matsuoka [21], pages 187–209.

[2] Kenichi Asai, Satoshi Matsuoka, and Akinori Yonezawa. Duplication and partial evaluation —for a better understanding of reflective languages—. *Lisp and Symbolic Computation*, 9:203–241, 1996.

[3] Lodewijk Bergmans and Mehmet Aksits. Composing crosscutting concerns using composition filters. *Communications of the ACM*, 44(10):51–57, October 2001.

[4] Anders Bondorf. Improving binding times without explicit CPS-conversion. In *ACM Conferenceon Lisp and Functional Programming*, pages 1–10, 1992.

[5] M. A. Bulyonkov. Polyvariant mixed computation for analyzer programs. *Acta Informatica*, 21:473–484, 1984.

[6] Yvonne Coady, Gregor Kiczales, Mike Feeley, Norm Hutchinson, and Joon Suan Ong. Structuring operating system aspects: using AOP to improve OS structure modularity. *Communications of the ACM*, 44(10):79–82, October 2001.

[7] Yvonne Coady, Gregor Kiczales, Mike Feeley, and Greg Smolyn. Using AspectC to improve the modularity of path-specific customization in operating system code. In *Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT symposium on Foundations of software engineering*, pages 88–98, Vienna, Austria, 2001.

[8] Rémi Douence, Olivier Motelet, and Mario Südholt. A formal definition of crosscuts. In Yonezawa and Matsuoka [21], pages 170–186.

[9] Yoshihiko Futamura. Partial evaluation of computation process—an approach to a compiler-compiler. *Higher-Order and Symbolic Computation*, 12(4):381–391, 1999. Reprinted from *Systems, Computers, Controls*, 2(5):45–50, 1971.

[10] Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, Cambridge, MA, 1991.

[11] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William Griswold. Getting started with AspectJ. *Communications of the ACM*, 44(10):59–65, October 2001.

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

[13] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *ECOOP '97 — Object-Oriented Programming 11th European Conference*, number 1241 in Lecture Notes in Computer Science, pages 220–242, Jyväskylä, Finland, 1997. Springer-Verlag.

[14] Hidehiko Masuhara, Satoshi Matsuoka, Kenichi Asai, and Akinori Yonezawa. Compiling away the meta-level in object-oriented concurrent reflective languages using partial evaluation. In Mary E. S. Loomis, editor, *Proceedings of Object-Oriented Programming Systems, Languages and Applications*, volume 30(10) of *ACM SIGPLAN Notices*, pages 300–315, Austin, TX, October 1995. ACM.

[15] Hidehiko Masuhara and Akinori Yonezawa. Design and partial evaluation of meta-objects for a concurrent reflective language. In Eric Jul, editor, *European Conference on Object-Oriented Programming (ECOOP'98)*, volume 1445 of *Lecture Notes in Computer Science*, pages 418–439, Brussels, Belgium, July 1998. Springer-Verlag.

[16] Doug Orleans and Karl Lieberherr. DJ: Dynamic adaptive programming in Java. In Yonezawa and Matsuoka [21], pages 73–80.

[17] Harold Ossher and Peri Tarr. Multi-dimensional separation of concerns using hyperspaces. Technical report, IBM, 1999.

[18] Brian Cantwell Smith. Reflection and semantics in Lisp. In *Conference record of Symposium on Principles of Programming Languages*, pages 23–35, 1984.

[19] Peter J. Thiemann. Cogen in six lines. In *International Conference on Functional Programming (ICFP'96)*, 1996.

[20] Mitchell Wand, Gregor Kiczales, and Christopher Dutchyn. A semantics for advice and dynamic joint points in aspect-oriented programming. In *Proceedings of The Ninth International Workshop on Foundations of Object-Oriented Languages (FOOL9)*, January 2002.

[21] Akinori Yonezawa and Satoshi Matsuoka, editors. *Third International Conference Reflection 2001*, volume 2192 of *Lecture Notes in Computer Science*, Koyoto, Japan, September 2001. Springer-Verlag.

A Formal Basis for Aspect-Oriented Specification with Superposition

Pertti Kellomäki, pk@cs.tut.fi
Institute of Software Systems
Tampere University of Technology
Finland

ABSTRACT

We present a formalization of how specifications are constructed using superposition and composition in the Ocsid specification language. The formalization covers stepwise refinement using superposition and composition of independent refinements. Independent views of a refinement hierarchy (subclassing and operation refinement) are reconciled in composition in a formally well founded way. The formalization also defines how classes and operations are constructed from fragments given in separate syntactical units.

The work has been done in the context of formal specification of distributed systems, but we believe the ideas to be useful in a more general setting as well.

1. INTRODUCTION

A prerequisite for effective separation of concerns is the ability to provide multiple views of a system being designed and to compose the views to form a coherent whole. We treat composition as two related but distinct activities: reconciling refinement hierarchies of the views, and determining the structure of entities populating the hierarchies. The first is concerned with relationships between entities, e.g. superclass–subclass, while the second is concerned with how entities are composed of fragments given in separate syntactic units.

Our work arises from formal specification of distributed systems using superposition, but we believe the ideas to be applicable in a more general setting as well. We wish to be able to formally verify temporal properties of specifications, so it is necessary to formally define how specifications are constructed and what the semantics of the resulting syntactic structures is. In this paper we formally define the semantics of superposition and composition in the Ocsid [12] specification language. The formalization defines the semantics in a very direct sense, as the definitions below are translitterations of the core of an Ocsid compiler written in the functional programming language Haskell.

Ocsid specifications are developed using stepwise refinement. Independent aspects of the system can be given in separate branches of specification, which may introduce subclasses and refinements of operations, and the branches may be merged in composition. Both stepwise refinement and composition preserve temporal safety properties.

The rest of the paper is structured as follows. In Section 2 we review superposition and the joint action approach to specification. Section 3 defines the notations used in the paper and Section 4 discusses derivation and extension. Section 5 contains the formalization of superposition and composition, Section 6 briefly discusses verification issues, and Section 7 gives a condensed example. Related work is reviewed and conclusions are drawn in Section 8.

2. SUPERPOSITION AND JOINT ACTIONS

Superposition is a well known technique for specifying distributed systems [8, 7, 5, 11, 6]. The specification language Ocsid described here is an experimental variant of the DisCo [9, 14, 2] specification language, both based on the use of superposition.

2.1 Superposition

Superposition in Ocsid relates to aspects as follows. Each superposition step describes a projection of the world relative to some set of state variables. The projection encapsulates a particular concern, which may crosscut several implementation components.

The result of superimposing a step on a base specification contains all the structure of the base specification, augmented with the additional structure given by the step. Successive applications of superposition result in a layered structure, so in accordance with the DisCo parlance we call superposition steps *layers*. The variant of superposition used in DisCo and Ocsid preserves temporal safety properties by construction by forbidding assignment to variables in the base specification.

A superposition step may include assumptions about the base specification, which facilitate verification of temporal properties of specifications resulting from applying the step. It is then sufficient to establish that the assumptions hold for a particular base specification to reuse the verification of the step.

2.2 Joint Actions

We give specifications using the *joint action* [3, 4] style. A joint action specification consists of a set of *classes*, a set of *joint actions*, and an initial condition. The state of the system is determined by *state variables* that reside in *objects*. An action has a set of *roles* in which objects may *participate*, and the action can be executed for any combination of objects for which the *guard* of the action evaluates to true. When an action is executed, the state of the participants is changed, while the rest of the system remains unchanged.

The formal semantics of joint action specifications has been given elsewhere [10], so we only outline it here. The semantics is given in terms of linear time temporal logic.

Each joint action gives rise to an existentially quantified formula that quantifies over the roles of the action. The

formula is formed of expressions in the guard of the action, expressions corresponding to assignments in the body, and conjuncts specifying that the rest of the world remains unchanged. Assignments map to logic as equalities that give the values of primed variables (next state) in terms of unprimed variables (current state).

The meaning of a syntactic specification is a temporal formula consisting of a conjunct corresponding to the initial condition, and a temporally quantified ("always") disjunction of the actions of the specification and the stuttering action. The stuttering action leaves all the variables of the specification unchanged.

3. NOTATION

The formalization makes heavy use of relations. We treat relations interchangeably as binary predicates and sets of pairs. The reflexive transitive closure of a relation $\mathcal{R}$ is denoted by $\mathcal{R}^*$.

We define a composition operation $\oplus$ for relations, which is monotonic in the sense that the composed relation may yield true for a pair for which neither component yields true, but it cannot yield false for a pair for which either one yields true. The composed relation $\mathcal{E}_1 \oplus \mathcal{E}_2$ yields true if either of the component relations yields true, or if they transitively yield true. The composition is defined as

$$\mathcal{E}_1 \oplus \mathcal{E}_2 \triangleq (\mathcal{E}_1 \cup \mathcal{E}_2)^*. \tag{1}$$

Rather than using the concrete syntax of the Ocsid language, we use the abstract syntax summarized in Table 1. We leave the more detailed syntactical elements underspecified, as they are not relevant to the present discussion. We use a sans serif font to indicate syntactic constructs in the definitions, and use terms *syntactic class* and *syntactic action* when emphasizing that we are referring to syntactic constructs.

class(*names*, *variables*)
action(*names*, *roles*, *guard conjuncts*, *assignments*)
specification(*class hierarchy*, *action hierarchy*, *initial conditions*, *class extensions*,*action extensions*)
layer(*classes*,*actions*, *initial conditions*, *class derivations*, *class extensions*, *action derivations action extensions*)
composition(*class mergings*,*action mergings*)

Table 1: Abstract syntax

4. DERIVATION AND EXTENSION

Our formalization is based on the notions of *derivation* and *extension*. The following discussion is applicable to both classes and actions, which we collectively call *entities*. Derivations tell what entities a specification contains and extensions determine the syntactic structure of the entities.

4.1 Derivation Hierarchies

A derivation hierarchy is a tuple

$$H = (N, \mathcal{E}, \mathcal{R}) \tag{2}$$

where N is a set of names and $\mathcal{E}$ and $\mathcal{R}$ are relations over the names.

Relation $\mathcal{E}$ is an equivalence relation that partitions the set of names into equivalence classes. Each equivalence class represents an entity in the specification.

Relation $\mathcal{R}$ records the derivation history in terms of names. The *is-a relation* $\mathcal{D}$ is defined as

$$\mathcal{D} \triangleq \mathcal{E} \oplus \mathcal{R}. \tag{3}$$

The notion of disjointness is useful for closed world modeling. The *disjointness relation* $\mathcal{S}$ describes which entities are disjoint from each other. Formally the disjointness relation is defined using $\mathcal{D}$ as

$$\mathcal{S}(a, b) \triangleq \neg\exists c : \mathcal{D}(c, a) \wedge \mathcal{D}(c, b). \tag{4}$$

4.2 Extension

An extension is a fragment of an entity. An entity in a specification is composed of the extensions that are applicable to the particular entity. A class extension contains state variables, and an action extension contains roles, guard conjuncts and assignments. Accessor functions $vars$, $roles$, $conjuncts$ and $assignments$ have their obvious definitions in the following. Function $name(e)$ returns the name of the entity to which extension e adds structure, and $names(n, H)$ returns the set (equivalence class) of names in N_H that are equivalent to n according to $\mathcal{E}_H$.

The syntactic structure of an entity is determined by a derivation hierarchy and a set of extensions. Function $structure_C$ returns the structure of the class denoted by name n relative to a hierarchy H and a set of class extensions X:

$$\begin{array}{l} structure_C(n, H, X) \triangleq \\ \quad \mathsf{class}\ (names(n, H), \bigcup_{x \in applicable(n,H,X)} vars(x)). \end{array} \tag{5}$$

Similarly, function $structure_A$ for actions is defined as

$$\begin{array}{l} structure_A(n, H, X) \triangleq \\ \quad \mathsf{action}\ (names(n, H), \\ \qquad \bigcup_{x \in applicable(n,H,X)} roles(x), \\ \qquad \bigcup_{x \in applicable(n,H,X)} conjuncts(x), \\ \qquad \bigcup_{x \in applicable(n,H,X)} assignments(x)). \end{array} \tag{6}$$

Set $applicable(n, H, X)$ is the set of extensions in X that are applicable when constructing the entity named n:

$$applicable(n, H, X) \triangleq \{x \in X \mid \mathcal{D}_H(n, name(x))\}. \tag{7}$$

The set consists of all the extensions that name an entity from which the entity named by n has been derived.

5. SPECIFICATIONS, SUPERPOSITION AND COMPOSITION

A specification is a tuple

$$S = \mathsf{specification}(H^c, H^a, I, X^c, X^a), \tag{8}$$

where H^c is a derivation hierarchy of class names, H^a is a derivation hierarchy of action names, I is a set of initial conditions, and X^c and X^a are sets of class and action extensions respectively. The structure of a class (resp. action) relative to the specification is determined by the hierarchy and extensions of the specification as explained above. Functions *class* and *action* return the syntactic structure corre-

sponding to a name n in specification S:

$$class(n,S) \triangleq structure_C(n, H_S^c, X_S^c) \quad (9)$$

$$action(n,S) \triangleq structure_A(n, H_S^a, X_S^a). \quad (10)$$

A superposition step is a tuple

$$L = \mathsf{layer}(C, A, I, D^c, X^c, D^a, X^a), \quad (11)$$

where C is a set of syntactic classes, A is a set of syntactic actions, I is a set of initial conditions, D^c is a set of class derivation pairs, X^c is a set of class extensions, D^a is a set of action derivation pairs, and X^a is a set of action extensions. A derivation pair is a tuple $(derived, base)$ denoting that *derived* has been derived from *base*.

A superposition step is well formed if the following conditions hold. The conditions are mostly trivial but tedious to formalize, so we list them here informally.

- Derivation pairs of the step only derive entities in C and A respectively or entities introduced by the step.
- Extensions of the step only extend entities in C and A, or entities introduced by the step.
- Expressions in an action extension only refer to roles present in A or roles introduced by the extension, and only to variables present in C or introduced in the step.
- Assignments in the action extensions only assign to variables introduced in the step.

A composition is a tuple

$$C = \mathsf{composition}(M^c, M^a), \quad (12)$$

where M^c and M^a are sets of sets of names denoting entities (classes and actions respectively) to be merged in the composition. A composition is well formed if the sets in both M^c and M^a are disjoint from each other.

Well-formedness is transitive: superimposing a well formed step on a well formed specification or composing a set of well formed specifications results in a well formed specification, provided that the side conditions are satisfied.

5.1 Superposition

Superimposing a set D of derivation pairs on a hierarchy $H = (N, \mathcal{E}, \mathcal{R})$ is defined as

$$superimpose(D, H) \triangleq (N', \mathcal{E}, \mathcal{R}') \quad (13)$$

where

$$\begin{aligned} N' &= N_H \cup \{d \mid (d,b) \in D\} \\ \mathcal{R}' &= \mathcal{R}_H \cup D. \end{aligned}$$

Superimposing a step L on a specification S is defined as

$$\begin{array}{l} superimpose(S, L) \triangleq \\ \quad \mathsf{specification}(\\ \quad\quad superimpose(D_L^c, H_S^c), \\ \quad\quad superimpose(D_L^a, H_S^a), \\ \quad\quad I_S \cup I_L, X_S^c \cup X_L^c, X_S^a \cup X_L^a). \end{array} \quad (14)$$

The result is a well formed Ocsid specification if the following side conditions hold.

- S and L are well formed.
- S contains a set C' of classes and a set of actions A' such that each element of C_L (resp. A_L) has a syntactically refined counterpart in C' (resp. A'). For a definition of syntactical refinement see below.
- Derivations and extensions do not introduce name conflicts.

A class $\widehat{C}$ syntactically refines a class C with name n, if n belongs to the names of $\widehat{C}$, and $\widehat{C}$ contains all the state variables of C. An action $\widehat{A}$ syntactically refines an action A with name n, if n belongs to the names of $\widehat{A}$, and $\widehat{A}$ contains all the roles of A.

Syntactic refinement ensures that there are no "dangling" references in the resulting specification. Stronger conditions are needed for independent verification of superposition steps, as explained in Section 6.

5.2 Composition

Composing a set $\mathcal{H}$ of hierarchies using a merging M is defined as

$$compose(\mathcal{H}, M) \triangleq (N', \mathcal{E}', \mathcal{R}') \quad (15)$$

where

$$\begin{aligned} N' &= \left(\bigcup_{m \in M} m\right) \cup \left(\bigcup_{h \in \mathcal{H}} N_h\right) \\ \mathcal{E}' &= buildMergeR(M) \oplus \left(\bigoplus_{h \in \mathcal{H}} \mathcal{E}_h\right) \\ \mathcal{R}' &= \bigcup_{h \in \mathcal{H}} \mathcal{R}_h. \end{aligned}$$

Function $buildMergeR$ takes a set of sets of names, and returns an equivalence relation. Each set of names indicates the names of entities to be merged in the composition. The function is defined as

$$buildMergeR(M) \triangleq \left(\bigcup_{m \in M} m \times m\right)^* \quad (16)$$

where $m \times m$ is the Cartesian product of m with itself.

The result of composition is well formed if the following side conditions hold ($\mathcal{D}'$ denotes the *is-a* relation of the result):

$$\forall h \in \mathcal{H} : \neg\exists n_1, n_2 \in N' : \mathcal{S}_h(n_1, n_2) \wedge \mathcal{D}'(n_1, n_2) \quad (17)$$

$$\begin{array}{l} \forall h \in \mathcal{H} : \\ \quad \neg\exists n_1, n_2 \in N' : \\ \quad\quad \mathcal{D}_h(n_1, n_2) \wedge \neg\mathcal{E}_h(n_1, n_2) \wedge \mathcal{E}'(n_1, n_2). \end{array} \quad (18)$$

The first side condition ensures that names that denote separate entities in any of the components cannot denote the same entity in the composed specification. The second side condition ensures that names that have an *is-a* relationship but do not denote the same entity in some component cannot denote the same entity in the result.

Composing a set $\mathcal{S}$ of specifications using a composition $C = \mathsf{composition}(M^c, M^a)$ is defined as

$$\begin{array}{l} compose(\mathcal{S}, C) \triangleq \\ \quad \mathsf{specification}(H^{c\prime}, H^{a\prime}, I', X^{c\prime}, X^{a\prime}) \end{array} \quad (19)$$

where

$$
\begin{aligned}
H^{c\prime} &= compose(\{H^c_s | s \in \mathcal{S}\}, M^c) \\
H^{a\prime} &= compose(\{H^a_s | s \in \mathcal{S}\}, M^a) \\
I' &= \bigcup_{s \in \mathcal{S}} I_s \\
X^{c\prime} &= \bigcup_{s \in \mathcal{S}} X^c_s \\
X^{a\prime} &= \bigcup_{s \in \mathcal{S}} X^a_s
\end{aligned}
$$

The result of composition is well formed if the following side conditions hold.

- Compositions of the hierarchies are well formed.
- Extensions do not create name conflicts.

6. VERIFICATION AND ABSTRACT STEPS

One of our main goals is to be able to construct formally verified specifications of distributed systems. Verification of individual specifications is fairly easy but not very attractive in practice, because it is difficult to reuse verification across different systems.

As noted earlier, a superposition step can be verified independently of its deployment if assumptions are made about the base specifications on which the step is to be superimposed. We use the following assumptions when verifying a superposition step $L = \mathsf{layer}(C, A, I, D^c, X^c, D^a, X^a)$.

- For each action $\mathcal{A}_B$ in the base specification corresponding to an action $\mathcal{A}_L$ in A, $\mathcal{A}_B \implies \mathcal{A}_L$ for all possible combinations of participants. An action is a predicate logic formula consisting of constants and references to unprimed and primed variables. Implication between actions is implication between the formulas.
- All actions in the base specification with no counterpart in A are stuttering with respect to state variables in C, i.e. they do not change the variables.

These assumptions are sufficient for verifying temporal safety properties of specifications resulting from superimposing L on a base specification. When the step is applied, it is sufficient to establish that the assumptions hold for the base specification.

A superposition step can be further abstracted by observing that proofs of temporal properties are insensitive to systematic renaming of classes, variables and actions. A relatively simple renaming mechanism can be used for adjusting a superposition step to a base specification in such a way that verification of temporal properties remains valid [12]. This facilitates the use of superposition steps as reusable templates.

7. EXAMPLE

This section presents a brief example using the concrete syntax of Ocsid. Figure 1 gives an Ocsid specification and two superposition steps (layers). The specification describes a system where nodes are arranged in a list using state variables *NEXT*. Action *delete* removes a node from a list by synchronously modifying two nodes.

Layer *request_reply* specifies how an atomic action of two participants modifying state variable V is implemented using a request message and a reply message. The requester of the operation sends a message and marks that its copy of the value (held in implementation-level variable v) may temporarily be invalid. The layer ensures that the value of state variable V can always be computed from other state variables, and consequently the synchronization implied by action A is not needed, in other words the synchronization is implemented by the message exhange.

Layer *token_passing* describes how an activity leading to execution of action A is coordinated using a token. The safety property ensured by the layer is that the execution of actions *reserve_token* and A a strictly alternates.

A specification of distributed removal coordinated with a token can be derived as follows. Simple substitution of *delete* for A, *node* for C etc. makes the layers compatible with specification *distlist*, and two separate specifications are created by using the layers so obtained. A third specification is then created as a composition of the two specifications. Classes *request* and *reserved_token* are merged to a single class, so a single object plays two specification level roles in the composed specification. Actions to send a request and to send a reserved token are composed (synchronized) to a single action in the composed specification.

According to the definitions of superposition and composition given earlier, the class known by names *request* and *reserved_token* is constructed using extension for both classes. The action to send a combined request and token is constructed similarly.

8. CONCLUSIONS

We have presented a formalization of how refinement hierarchies are composed in the Ocsid specification language, and how the syntactic structure of classes and actions is determined. Composition of refinement hierarchies is equivalent to reconciliation of different views on a system, while determining the syntactic structure of entities is concerned with how entities are composed of fragments given in separate syntactic units.

There are obvious parallels between our work and that of Tarr et al. [16] on composing separate views. Our composition is much more restricted than theirs, because we wish to preserve temporal safety properties in composition. The overall effect of composition in Ocsid is the same as that of weaving e.g. as in AspectJ [13, 1]. One can think of class and action names as explicit join points, but the analogy is somewhat stretched as in Ocsid there is no base structure into which to weave aspects.

Superposition is a well established methodology for the specification of distributed systems. Our work is closest to work on the DisCo specification language [9, 14, 2], but it is also very similar to the use of superposition in Unity [6]. Independent verification of superposition steps follows the ideas of Katz in [11].

Our formalization is not just an idle exercise in formal mathematics, but an integral part of a compiler for the Ocsid language. The compiler is written in the functional language Haskell, and the definitions in the compiler mirror those given in this paper.

We plan to use the formalization for a deep embedding of the Ocsid language into the logic of the PVS [15] theorem prover. A deep embedding would enable us to verify prop-

```
specification distlist is
    class node is NEXT : ref node; end;
    action delete by n1, n2 : node
    when n1.NEXT = ref(n2)
    do n1.NEXT := n2.NEXT; n2.NEXT := null; end;
end;
layer request_reply requires
    type T; class C is V : T; end;
    action A by p1, p2 : C when true
    do p1.V := _; p2.V := _; end;
provides
    class request; class reply;
    class extension request is from : ref C; v : T; end;
    class extension reply is to : ref C; v : T; end;
    class extension C is v : T; valid : boolean; end;
    actions to send and receive messages omitted
    action extension A by ... req : request; rep : reply
    when ... req.from = ref(p1) and rep.to = null
        and p2.valid
    do ... rep.to := req.from; rep.v := 'p1.V;
        req.from := null; p2.v := 'p2.V;
    end;
end;
layer token_passing requires
    class C;
    action A by p : C is when true do end;
provides
    class free_token; class reserved_token;
    class extension free_token is at : ref C; end;
    class extension reserved_token is at : ref C; end;
    action reserve_token;
    actions to pass tokens around omitted
    action extension reserve_token
    by ... p : C; rt : reserved_token; ft : free_token
    when ... ft.at = ref(p)
    do ... ft.at := null; rt.at := ref(p); end;
    action extension A
    by ... rt : reserved_token; ft : free_token
    when ... rt.at = ref(p)
    do ... rt.at := null; ft.at := ref(p); end;
end;
```

Figure 1: An example in Ocsid concrete syntax.

erties of specifications written in the Ocsid language as well as properties of the language itself.

Acknowledgments

This research was partly supported by Academy of Finland, project ABESIS grant number 5100005. Discussions of an early draft of this paper with professor Markku Sakkinen were extremely helpful in simplifying the formalization.

9. REFERENCES

[1] The AspectJ home page. At URL http://www.aspectj.org/, 2002.

[2] The DisCo project WWW page. At `http://disco.cs.tut.fi` on the World Wide Web, 2001.

[3] R. J. R. Back and R. Kurki-Suonio. Distributed cooperation with action systems. *ACM Transactions on Programming Languages and Systems*, 10(4):513–554, October 1988.

[4] R. J. R. Back and R. Kurki-Suonio. Decentralization of process nets with a centralized control. *Distributed Computing*, (3):73–87, 1989.

[5] K. M. Chandy and L. Lamport. Distributed snapshots: determining the global state of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, 1985.

[6] K. M. Chandy and J. Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, 1988.

[7] K. Mani Chandy, Jayadev Misra, and Laura M. Haas. Distributed deadlock detection. *ACM Transactions on Computer Systems*, 1(2):144–156, May 1983.

[8] Edsger W. Dijkstra and C. S. Scholten. Termination detection for diffusing computations. *Information Processing Letters*, 11(1):1–4, August 1980.

[9] H.-M. Järvinen, R. Kurki-Suonio, M. Sakkinen, and K. Systä. Object-oriented specification of reactive systems. In *Proceedings of the 12th International Conference on Software Engineering*, pages 63–71. IEEE Computer Society Press, 1990.

[10] Hannu-Matti Järvinen. *The design of a specification language for reactive systems*. PhD thesis, Tampere University of Technology, 1992.

[11] Shmuel Katz. A superimposition control construct for distributed systems. *ACM Transactions on Programming Languages and Systems*, 15(2):337–356, April 1993.

[12] Pertti Kellomäki. A structural embedding of Ocsid in PVS. In Richard J. Boulton and Paul B. Jackson, editors, *Theorem Proving in Higher Order Logics, TPHOLS2001*, number 2152 in Lecture Notes in Computer Science, pages 281–296. Springer Verlag, 2001.

[13] Gregor Kiczales, Jim Hugunin, Mik Kersten, John Lamping, Cristina Lopes, and William G. Griswold. Semantics-Based Crosscutting in AspectJ. In *Workshop on Multi-Dimensional Separation of Concerns in Software Engineering (ICSE 2000)*, 2000.

[14] Reino Kurki-Suonio. Fundamentals of object-oriented specification and modeling of collective behaviors. In H. Kilov and W. Harvey, editors, *Object-Oriented Behavioral Specifications*, pages 101–120. Kluwer Academic Publishers, 1996.

[15] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752. Springer Verlag, 1992.

[16] Peri Tarr, Harold Ossher, William Harrison, and Stanley M. Sutton, Jr. *N* degrees of separation: Multi-dimensional separation of concerns. In *Proceedings of the 1999 International Conference on Software Engineering*, pages 107–119. IEEE Computer Society Press / ACM Press, 1999.

Observers and Assistants: A Proposal for Modular Aspect-Oriented Reasoning

Curtis Clifton and Gary T. Leavens
Department of Computer Science
Iowa State University
226 Atanasoff Hall
Ames, IA 50011-1040 USA
+1 515 294 1580
{cclifton,leavens}@cs.iastate.edu

ABSTRACT

In general, aspect-oriented programs require a whole-program analysis to understand the semantics of a single method invocation. This property can make reasoning difficult, impeding maintenance efforts, contrary to a stated goal of aspect-oriented programming. We propose some simple modifications to AspectJ that permit modular reasoning. This eliminates the need for whole-program analysis and makes code easier to understand and maintain.

1. INTRODUCTION

Much of the work on aspect-oriented programming languages makes reference to the work of Parnas [23]. That work argues that the modules into which a system is decomposed should be chosen to provide benefits in three areas. Parnas writes (p. 1054):

> "The benefits expected of modular programming are: (1) managerial—development time should be shortened because separate groups would work on each module with little need for communication; (2) product flexibility—it should be possible to make drastic changes to one module without a need to change others; (3) comprehensibility—it should be possible to study the system one module at a time. The whole system can therefore be better designed because it is better understood."

While much has been written about aspect-oriented programming as it relates to Parnas's second point, his third point is the primary concern of this paper. We contend that current aspect-oriented programming languages do not provide this third benefit in general, because they require systems to be studied in their entirety.

After describing and motivating the problem in this introduction, in Section 2 we propose a simple set of restrictions that, we believe, would bring these languages much closer to Parnas's ideal without any practical loss of expressiveness. This proposal is preliminary work and is presented in the hopes of generating discussion and feedback.

We begin by defining a notion of modular reasoning corresponding to Parnas's third benefit. Subsequent subsections in this introduction show how such modular reasoning is possible in the Java Programming Language™ [1, 9] but problematic in the current version of AspectJ™ [11].

For concreteness, our examples are shown in AspectJ. To support abstract reasoning we specify the examples using new aspect-oriented extensions to the Java Modeling Language (JML) [13, 14]. We believe our ideas are independent of Java and JML. We also believe that they are independent of the details of AspectJ and are generally applicable to the class of aspect-oriented languages.

1.1 Modular Reasoning

Before delving into the details, it is useful to define our terms. *Modular reasoning* is the process of understanding a system one module at a time. A language *supports modular reasoning* if the actions of a module *M* written in that language can be understood based solely on the code contained in *M* along with the specifications of any modules referred to by *M*. For example, in Java a single compilation unit, typically a file declaring a single top-level type (class or interface), is a module. The specification of that module is the behavior of objects of that type. Code is one form of specification. In a more expressive language, such as Eiffel [19] or Java annotated with JML, a specification can be given explicitly using pre- and postconditions, frame axioms, and invariants; such specifications serve as contracts that allow one to separately reason about the behavior and correctness of an implementation.

Our interest in modular reasoning in aspect-oriented programming languages is motivated in part by our earlier work on MultiJava [5, 6]. In that work we were concerned with modular static typechecking and compilation. This is closely related to the issue of modular reasoning, because the source code of a method body is a very precise behavioral specification of that method. A language that supports modular reasoning can therefore also permit separate compilation, as well as modular implementations of other tools (e.g., optimizers, verifiers, and model checkers). Thus, mechanisms that permit modular reasoning would have many benefits.

1.2 Modular Reasoning in Java

Java without aspect-oriented extensions supports modular reasoning. We illustrate this after giving some background on JML.

1.2.1. JML Background

Consider the examples in Figure 1 and Figure 2, modified slightly from Kiczales, *et al.* [11] and annotated with JML specifications. Specification annotations are enclosed in special comments; at-signs (`@`) at the beginning of lines in annotations are ignored.

In JML, *model fields*, like `xCtr` and `yCtr` in Figure 1, specify the abstract state of an object. They are specification-only constructs, but are treated formally as locations. The keyword `instance` says that they are considered to be model fields in all classes that implement the interface. A represents-clause (with keyword `represents`, as in Figure 2) says how the values of model fields are related to the actual, concrete fields of an object; and a depends-clause (also in Figure 2) allows such concrete fields to be assigned to when the model fields that depend on them are assignable [15].

In our JML examples, each method's specification is written preceding its signature. We use a desugared form of JML method specifications in this paper, in which a visibility level, which describes who can see the specification, is followed by the keyword `behavior`, which introduces a *specification case*. A specification case consists of several clauses. The forall-clause introduces logical

```
package foal02;
interface FigureElement {
   /*@ model instance int xCtr, yCtr; @*/
   /*@ public behavior
     @  forall int oldx, oldy;
     @  requires oldx == xCtr && oldy == yCtr
     @         && dx >= 0 && dy >= 0;
     @  assignable xCtr, yCtr;
     @  ensures xCtr == oldx + dx
     @      && yCtr == oldy + dy
     @      && \result == this;
     @  signals (Exception z) false; @*/
   FigureElement moveNE( int dx, int dy );
   /*@ public behavior
     @   ensures \result == xCtr;
     @   signals (Exception z) false; @*/
   /*@ pure @*/ int getX();
   /*@ public behavior
     @   ensures \result == yCtr;
     @   signals (Exception z) false; @*/
   /*@ pure @*/ int getY();
}
```

Figure 1: A Java module declaring an interface, with (unsugared) JML specifications

```
package foal02;
class Point implements FigureElement {
   private int _x = 0, _y = 0;
   /*@ private depends xCtr <- _x;
     @ private represents xCtr <- _x; @*/
   /*@ private depends yCtr <- _y;
     @ private represents yCtr <- _y; @*/
   /*@ public behavior
     @   assignable xCtr, yCtr;
     @   ensures xCtr == x && yCtr == y;
     @   signals (Exception z) false; @*/
   public Point( int x, int y ) {
      _x = x;  _y = y;
   }
   public /*@ pure @*/ int getX() { return _x; }
   public /*@ pure @*/ int getY() { return _y; }
   /*@ public behavior
     @   assignable xCtr;
     @   ensures xCtr == x;
     @   signals (Exception z) false; @*/
   public FigureElement setX( int x ) {
      _x = x;
   }
   /*@ public behavior
     @   assignable yCtr;
     @   ensures yCtr == y;
     @   signals (Exception z) false; @*/
   public FigureElement setY( int y ) {
      _y = y;
   }
   /*@ also
     @  public behavior
     @   requires dx < 0 || dy < 0;
     @   ensures false;
     @   signals (Exception z)
     @     z instanceof IllegalArgException;
     @*/
   public FigureElement moveNE( int dx, int dy ) {
      if (dx < 0 || dy < 0) {
         throw new IllegalArgException();
      }
      setX( getX() + dx );
      setY( getY() + dy );
   }
}
```

Figure 2: A Java module declaring a class

variables that are universally quantified over the specification case. The requires-clause gives the case's precondition, the assignable-clause its frame, the ensures-clause its normal postcondition, and the signals-clause its exceptional postcondition. Postconditions may use the keyword `\result` to refer to the value a method returns. Consider a call to the method being specified; for all assignments to the universally quantified variables that make the precondition true, the call may only mutate locations described by the frame, and if the call returns normally the method must satisfy the normal postcondition; if the call throws an exception it must satisfy the exceptional postcondition. For brevity we will omit empty forall-clauses, requires-clauses with the default predicate `true`, and assignable-clauses for which the frame has the default value of `\nothing`.

The form of method specifications we use in this paper is not the one users normally write in JML. But it is useful for our semantic study because it corresponds directly to the semantics. For example, JML borrows from Eiffel the ability to refer to the pre-state value of an expression *E* in a postcondition by writing `\old`(*E*). The desugaring we assume here is to bind the pre-state values of each variable referred to in a `\old` expression to a fresh logical variable introduced in a forall-clause. JML also permits the use of calls to "pure" (side-effect free) methods in specifications, but in this paper we do not consider such calls. Instead we assume that calls to such methods are interpreted using the logical formulas in their normal postconditions. For example, one would normally write the specification of `moveNE` in Figure 1 as follows.

```
/*@ public behavior
  @  requires dx >= 0 && dy >= 0;
  @  assignable xCtr, yCtr;
  @  ensures getX() == \old(getX() + dx)
  @      && getY() == \old(getY() + dy)
  @      && \result == this;
  @  signals (Exception z) false;*/
FigureElement moveNE( int dx, int dy );
```

1.2.2. Java Examples

Suppose one wanted to write code that manipulates objects of type `FigureElement`. One could reason about such objects based solely on the information contained in Figure 1. That is, one would know the objects support a method named `moveNE` that takes two arguments of type `int` and that both arguments must be non-negative. Also if this precondition is satisfied, then the method will leave the object in a state where the values returned by `getX` and `getY` were increased by `dx` and `dy`, respectively.

Similarly, suppose one wanted to write code that manipulated instances of `Point`. One could reason about these instances based on Figure 2, along with the modules referred to in that code. To reason about `Point`'s `moveNE` method we would need to consider the specification of the `FigureElement` module since (in JML) methods inherit the specifications of the methods that they override and the method signatures that they implement. But this consideration of `FigureElement` is still modular because `FigureElement` is explicitly referred to by the clause

```
implements FigureElement
```

in the declaration of `Point`.[1] The additional specification for `moveNE` in the `Point` module is combined with the inherited specification from `FigureElement` to form the *effective specification* (i.e., the complete specification that must be satisfied at run-time)

1. In Java every class is implicitly a subclass of `java.lang.Object`. Thus reasoning in Java also requires that one consider `Object`'s specification. However, because it is common to all classes we do not consider this implicit reference to be non-modular.

```
package foal02;
aspect PointMoveChecking {
   private final String MOVE_SW =
      "did you mean to call moveSW()?";
   before(Point p, int dx, int dy):
      target(p) && args(dx, dy)
         && call(FigureElement moveNE(int,int))
      {
         if (dx < 0 && dy < 0)
            throw new
               IllegalArgException(MOVE_SW);
      }
}
```

Figure 3: An AspectJ module providing advice for `Point`

of `Point`'s `moveNE` method. Rules for combining inherited specifications in JML give the following effective specification for `Point`'s `moveNE` method [24]:

```
 public behavior
   forall int oldx, oldy;
   requires oldx == xCtr && oldy == yCtr
         && dx >= 0 && dy >= 0;
   assignable xCtr, yCtr;
   ensures xCtr == oldx + dx
         && yCtr == oldy + dy
         && \result == this;
   signals (Exception z) false;
also
 public behavior
   requires dx < 0 || dy < 0;
   ensures false;
   signals (Exception z)
         z instanceof IllegalArgException;
```

JML's `also` keyword combines specification cases; it says that when the precondition of one of the combined cases holds, then the rest of that specification case must be satisfied. So, in addition to the specification inherited from `FigureElement`, this effective specification says that when a client fails to satisfy the original precondition the implementation must throw an `IllegalArgException`. (This inheritance enforces behavioral subtyping [7, 18].)

1.3 Non-modular Reasoning in AspectJ

Next we show that modular reasoning is not a general property of AspectJ by considering an aspect-oriented extension to our previous example. Figure 3 gives an aspect, `PointMoveChecking`, that modifies the behavior of `Point`'s `moveNE` method. `PointMoveChecking` declares a piece of *before-advice*, or code to be executed before traversing a join point into a method body. A *join point* is an arc in the dynamic call graph of a program.[2] The before-advice in `PointMoveChecking` is applicable to each join point where a target object of type `Point` receives a call to the method with signature `FigureElement moveNE(int,int)`. The `target` and `args` keywords are used to give names to the target object and arguments of the method call. (AspectJ also has *after-advice*, executed after traversing a join point out of a method body, and *around-advice*, which applies to the join points into and out of a method body.)

The before-advice in `PointMoveChecking` throws an exception if the absolute value of both arguments in a call to `Point`'s `moveNE` method is less than 0. In AspectJ this advice is applied by the compiler without explicit reference to the aspect from either the `Point` module or a client module; so by definition, modular reasoning about the `Point` module or a client module does not consider the `PointMoveChecking` aspect. Thus, modular reasoning has no way to detect that the effective specification of the `moveNE` method should be changed when the `Point` module and `PointMoveChecking` are compiled together. However, when they are compiled together, then intuitively the behavior of `Point`'s `moveNE` method satisfies the following specification[3].

```
 public behavior
   forall int oldx, oldy;
   requires oldx == xCtr && oldy == yCtr
         && dx >= 0 && dy >= 0;
   assignable xCtr, yCtr;
   ensures xCtr == oldx + dx
         && yCtr == oldy + dy
         && \result == this;
   signals (Exception z) false;
also
 public behavior
   requires dx < 0 || dy < 0;
   ensures false;
   signals (Exception z)
      z instanceof IllegalArgException;
also
 public behavior
   requires dx < 0 && dy < 0;
   ensures false;
   signals (Exception z)
      z instanceof IllegalArgException
      && z.getMessage().equals(MOVE_SW);
```

Unfortunately, this behavior is only available to the programmer via non-modular reasoning. That is, in AspectJ the programmer must potentially consider every aspect that refers to the `Point` class in order to reason about the `Point` module. So, in general, a programmer cannot "study the system one module at a time" [23].

1.4 Problem Summary

In a paper from ECOOP 2001, arguing for aspect-oriented programming, Kiczales, *et al.* state [12] (p. 327):

> "We would like the modularity of a system to reflect the way 'we want to think about it' rather than the way the language or other tools force us to think about it."

However, we have seen that the lack of support for modular reasoning can sometimes prevent us from thinking about a system "the way we want to think about it". In AspectJ, tool support is provided to compensate for this lack of modularity. Such tools perform the necessary whole program analysis to direct the programmer to the applicable aspects that affect pieces of a module's source code. Other tools for processing AspectJ source code (e.g., typecheckers, compilers, and optimizers) also require a whole program analysis.

We seek a small set of modifications to AspectJ that obviate the need for this whole program analysis either by the programmer or by supporting tools.

The remainder of this paper is organized as follows. Section 2 gives our proposal for modular reasoning. Section 3 evaluates our proposal. Section 4 discusses some limitations of our proposal and considers separate compilation. Section 5 concludes.

2. A PROPOSAL

We have shown that AspectJ in general does not support modular reasoning; in general the effective specification of a module can only be determined by a whole-program analysis. In this section we propose modifying AspectJ by categorizing aspects into two sorts: assistants and observers. "Observers" are limited in that they may not change the effective specifications of the modules they apply to, "assistants" are not limited in this way. Since observers do not change effective specifications, they preserve modular reasoning even when applied without explicit reference by the modules they

2. Join points in AspectJ are actually more general than what we describe. For example, join points can refer to field references and exception handlers [2]. We leave generalization for future work.

3. We will formalize this intuition in Section 2.

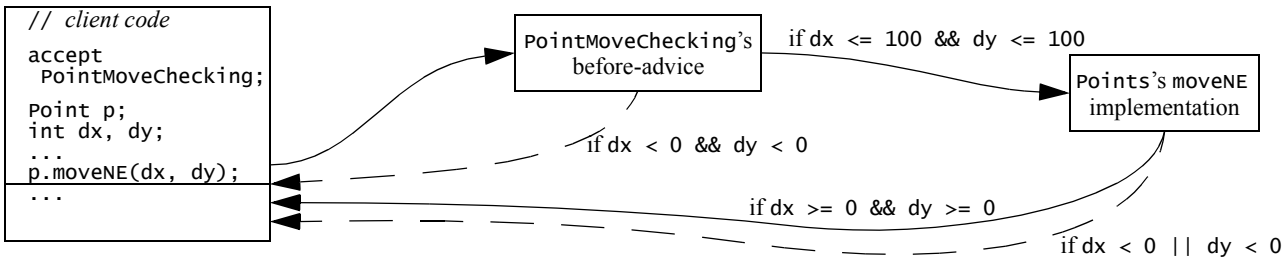

Figure 4: A depiction of the possible control flows of invocations of `moveNE` given that the client module accepts `PointMoveChecking`'s assistance (dashed lines represent exceptional control flow)

observe. Hence observers preserve most of the flexibility of the current version of AspectJ. Because assistants can change the effective specification of the modules to which they apply, to maintain modular reasoning they can only be applied in modules that explicitly reference them. Assistants also require subtle reasoning techniques.

2.1 Assistants

We call aspects that can change the effective specification of a module *assistants*. The `PointMoveChecking` aspect of Figure 3 is an assistant. The term "assistant" is intended to connote a participatory role for these aspects.

What information is needed to modularly reason about behavior when assistants are present? Quite simply, a module must explicitly name those assistants that may change its effective specification or the effective specifications of modules that it uses. We say that a module *accepts assistance* when it names the assistants that are allowed to change its effective specification or the effective specifications of modules that it uses. Assistance may be accepted by:

- the module to which the assistance applies, or
- a client of that module.

AspectJ does not currently include syntax for explicitly accepting assistance. We propose a simple syntax extension for this purpose:

```
accept TypeName;
```

where *TypeName* must be a canonical name of an assistant, i.e., a fully qualified name of the package containing the assistant, followed by a "dot" (`.`), followed by the assistant's identifier. Multiple accept-clauses may appear in a single module, following any import-clauses. For example, the `Point` module could accept the `PointMoveChecking` assistance by declaring:

```
accept foal02.PointMoveChecking;
```

Since `Point` (the module implementing the `moveNE` method) accepts `PointMoveChecking`'s assistance, this assistance is applied to every call to `Point`'s `moveNE` method, regardless of the client making the call.

On the other hand, if `PointMoveChecking`'s assistance was accepted by a client module, then that assistance would only be applied to calls from that client to `Point`'s `moveNE` method. Other clients that did not accept the assistance would not have it applied to their calls.

Figure 4 depicts the control flow of an invocation of `moveNE` with `PointMoveChecking`'s assistance. This depiction shows that there are multiple paths by which control may return to the client code, depending on the values of the parameters. The two arrows from the implementation of `moveNE` back to the client code correspond to the two postconditions (normal and exceptional) in the method's specification. Dashed lines indicate exceptional control flow.

2.1.1. Composing Advice Specifications

When a client invokes a method for which either the client or

```
package foal02;

aspect PointMoveChecking {

   private final String MOVE_SW =
      "did you mean to call moveSW()?";

   /*@  public behavior
     @   requires dx > 0 || dy > 100;
     @   ensures true;
     @   signals (Exception z) false;
     @ also
     @  public behavior
     @   requires dx < 0 && dy < 0;
     @   ensures false;
     @   signals (Exception z)
     @     z instanceof IllegalArgException
     @     && z.getMessage().equals(MOVE_SW); @*/
   before(Point p, int dx, int dy):
      target(p) && args(dx, dy)
         && call(FigureElement moveNE(int,int))
      {
         if (dx < 0 && dy < 0)
            throw new
                  IllegalArgException(MOVE_SW);
      }
}
```

Figure 5: Assistant from Figure 3 with specification added

implementation module has accepted assistance, the behavior of that invocation is based on the sequential composition of the code along a particular control flow path. We can reason abstractly about the possible behavior of the invocation by considering specifications for the method and the assistants. In this subsection we extend JML to specify advice in AspectJ and we show how to modularly reason about the effective specification of a method in the presence of accepted assistance.

A specification language for an aspect-oriented programming language must take possible control flow paths into account. Figure 5 gives another version of the `PointMoveChecking` assistant that adds a specification for the before-advice. In before-advice an ensures-clause gives a normal postcondition, which must hold before control passes to the advised method (or any other applicable advice). We use the ensures-clause in this way since passing control to the advised method is the "normal" behavior for before-advice. So the specification of the before-advice in Figure 5 says that if the advice is entered with `dx > 0` or `dy > 0`, then control flow must pass to the advised method. The implicit frame axiom for this case says that no relevant locations may be assigned when this precondition holds.

The second specification case in Figure 5 says that if the advice is entered with `dx < 0` and `dy < 0` then control flow must return to the caller by throwing an `IllegalArgException` whose message is "`did you mean to call moveSW()?`".

When reasoning about a call to `Point`'s `moveNE` method from the client's perspective we would like to use an effective specification that abstracts away the details of the control flow and intermediate

state transformations. That is, the effective specification from the client's perspective should just concern the preconditions as control flow leaves the client and the postconditions as control flow returns to the client, along with the relevant frame axioms.

Just as the effective behavior along any control flow path is the sequential composition of the code along that path, the effective specification along any control flow path is formed by a kind of sequential composition of the specifications along that path. When a set of paths are in parallel, as in our example, then the effective specification of the set is a kind of parallel composition of the parallel paths' specifications. To formalize these notions we will begin by just considering before-advice and after-returning advice.[4] Then we will use the model to determine the effective specification of `moveNE` in our running example. Later we will sketch extensions to our formal model to accommodate around-advice.

We present our model in two stages. We first describe how to construct a specification composition graph, from the specifications of the implementation module and those of any assistants accepted by that module or the client module. We then describe how the graph is used to determine the effective specification of the invocation.

Constructing a Specification Composition Graph

A *specification composition graph* is a graph whose vertices represent a single method specification, the specifications of all advice applicable to the method (and accepted by the method's implementation module or the client module), and the prestate and poststate from the client's view. The specification composition graph is analogous to the control flow graph for the corresponding code. The specification composition graph is used to determine the possible paths through the advice and method specifications (and hence the code if the implementation is correct). These paths are used to calculate the effective specification.

In general a module may accept assistance from multiple assistants and both a client and an implementation module may accept assistance. The specification composition graph is formed respecting the following order:

1. Apply any before-advice accepted by the client module in the order that it is accepted.
2. Apply any before-advice accepted by the implementation module in the order that it is accepted.
3. Execute the method body.
4. Apply any after-advice accepted by the implementation in the reverse order from which it is accepted.
5. Apply any after-advice accepted by the client module in the reverse order from which it is accepted.

This ordering ensures, for example, that the first assistance accepted by the client is "nearest" to the client and that the last assistance accepted by the implementation is nearest to the implementation on any path.

We will denote this ordering of before-advice, the method, and after-advice by the sequence $\langle a_1, a_2, \ldots, a_n \rangle$ where a_1 through a_{m-1} represent the before-advice, a_m represents the method, and a_{m+1} through a_n represents the after-advice.

4. AspectJ supports three kinds of after-advice. After-returning advice is only applicable when the advised method exits normally. After-throwing advice is only applicable when the advised method exits by throwing an exception. Regular after-advice, without a returning- or throwing-clause, is applicable in either case. To avoid complications in this preliminary proposal we are only considering after-returning advice. It is a simple matter to modify the edge construction algorithm, presented below, to accommodate the other kinds of after-advice.

(For simplicity and modularity we have decided for the present to confine acceptance of assistance to the module in which it is explicitly accepted. For example, if `ColorPoint` is a subclass of `Point`, assistance accepted by `Point` is not automatically applied to invocations of methods declared in `ColorPoint`. On the other hand, if for a particular method `ColorPoint` does not override `Point`'s implementation, then the inherited method carries with it the assistance accepted in the `Point` module. This approach also provides flexibility since the programmer can always add an accept-clause to the subclass module or override a superclass method (gaining assistance in the first case and "shadowing" assistance acceptance in the second). Similar considerations apply for assistance accepted by a superclass module of a client class. Also for simplicity we do not allow interfaces to accept assistance. Future work may reevaluate these decisions.)

Figure 6 gives the specification composition graph for `Point`'s `moveNE` method with assistance from `PointMoveChecking`. It is helpful to refer to this figure while considering the graph construction algorithm.

Formally, a specification composition graph is a directed acyclic graph, $G = \langle V, E \rangle$, where $V = \{start, end\} \cup \{a_i \mid 1 \le i \le n\}$ is the set of vertices and E is the set of edges.

As Figure 6 shows, each vertex in V, except *start* and *end*, is annotated with the signature of the corresponding method or advice. This information is used reason about the passing of parameters.

To define the edges of the specification composition graph, we first define a function *next* that orders the vertices. Let $next(start) = a_1$, for all $1 \le i \le n\text{-}1$, $next(a_i) = a_{i+1}$, and $next(a_n) = end$.

We also need some notation that will be used to label edges in the graph with information from the advice and method specifications. We will use Σ to represent the set of all possible program states, i.e., the set of all legal assignments of values to locations. For each a_i in V, let its specification, $S(a_i)$, be represented by a set of tuples, $S(a_i) = \{S_k(a_i) \mid S_k(a_i) = \langle Q_k, r_k, f_k, e_k, s_k \rangle, 1 \le k \le p_i\}$, where p_i is the number of cases in the specification and for all k:, $1 \le k \le p_i$, $S_k(a_i)$ represents the kth specification case, in which:

- Q_k represents its set of quantified variables (from `forall`), along with the implicitly bound `\result` variable for methods and after-advice and any variables bound in signals-clauses.
- $r_k : \Sigma \to \text{Bool}$ represents its precondition (`requires`),
- f_k is a set of variables that represents its frame (`assignable`),
- $e_k : \Sigma \to \text{Bool}$ represents its normal postcondition (`ensures`), and
- $s_k : \Sigma \to \text{Bool}$ represents its exceptional postcondition (`signals`).[5]

Each edge in E is represented by a tuple, $\langle \rho, x, y, S_k(x), \sigma \rangle$, with

- $\rho \in \{\nu, \varepsilon\}$ indicating normal (ν) or exceptional (ε) control flow,
- x and y being the beginning and ending vertices of the edge,
- $S_k(x)$ being the kth specification case (as above), and
- $\sigma \in \Sigma$ being the state of the program when control flow traverses that edge.

5. To avoid unnecessary additional complexity we assume each specification in this representation already includes the specifications inherited from its supertypes. Also, typically postconditions are modeled as relations on two states, but we are assuming a form for postconditions that cannot refer to pre-state values.

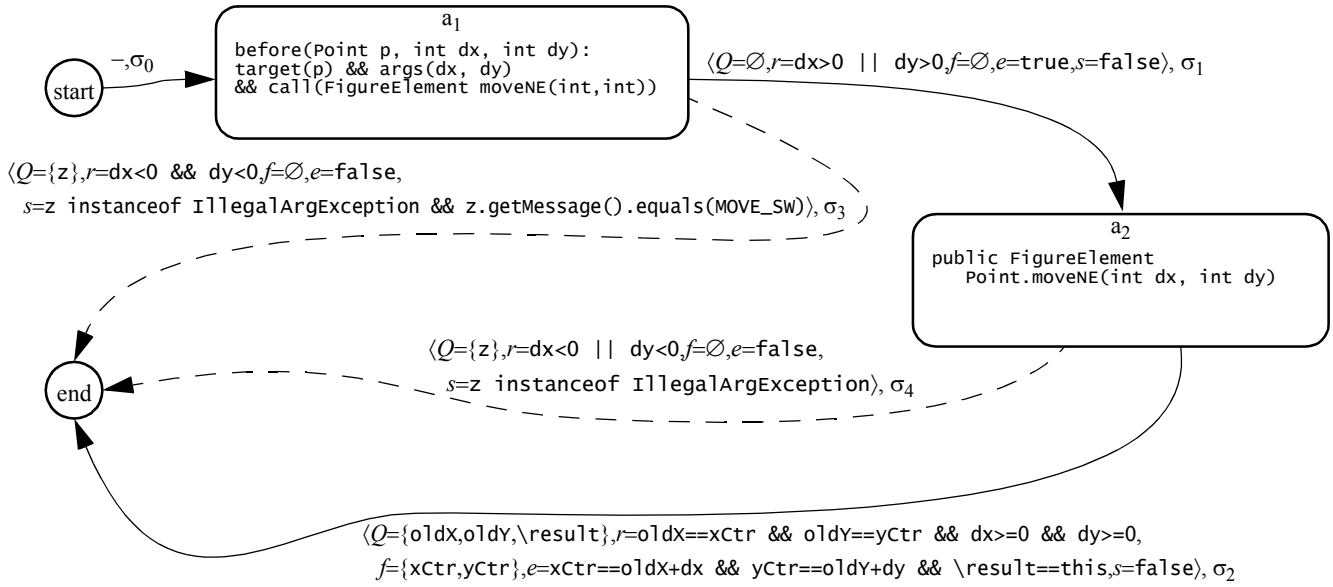

Figure 6: The specification composition graph for `Point`'s `moveNE` method with assistance accepted from `PointMoveChecking` Vertex a_1 corresponds to `PointMoveChecking`'s advice specification, vertex a_2 to `Point`'s `moveNE` specification; edges are labeled with the specification case of the start vertex and the name of the state; exceptional edges are shown with dashed lines.

To model all the possible normal and exceptional control paths, construct E as follows:

1. Let J be an index set of distinct indexes. These will be used to label the state information in the edge tuples.
2. Add a directed edge $\langle \nu, start, next(start), –, \sigma_j \rangle$ to E, where $j \in J$ is an unused index. The empty specification '–' is not used when computing the effective specification.
3. Let $x := next(start)$.
4. Repeat until $x = end$:
   4.1. For each specification case $S_k(x)$ in $S(x)$, if e_k is not `false`, add a *normal edge* $\langle \nu, x, next(x), S_k(x), \sigma_i \rangle$ to E and if s_k is not `false`, add an *exceptional edge* $\langle \varepsilon, x, end, S_k(x), \sigma_j \rangle$ to E, where $i, j \in J$ are unused indices.
   4.2. Let $x := next(x)$.

Figure 6 shows the specification composition graph generated by this algorithm when `Point` accepts assistance from `PointMoveChecking`;

Composing Specifications Along A Path

The specification composition graph, G, contains all the information needed to calculate the effective specification of a method invocation. We first describe how to compose specifications along any single path in G.

Consider a unique path from *start* to *end* in the graph. Because of exceptional return edges this path may not visit every node in the graph. For simplicity of notation we will sequentially renumber the states and for each a_i we will write S_i for the specification case from $S(a_i)$ used on this path. Thus, the path is:

$$\langle \langle \nu, start, a_1, –, \sigma_0 \rangle, \langle \rho, a_1, a_2, S_1, \sigma_1 \rangle, \ldots \\ \ldots, \langle \rho, a_q, end, S_q, \sigma_q \rangle \rangle,$$

where there are q+1 edges on the path. (For a path without exceptional edges $q = n$, otherwise a_q throws the exception.)

To prevent capture of the locally bound variables when composing the specifications, we α-convert the specification cases and related method and advice signatures so that all bound variable names are unique. We reserve the method's formal parameter names for prestate values, so we must α-convert the signature and out-edges of the method vertex. We also reserve the `\result` keyword for the poststate of the effective specification and so all instances of \result in the graph must be α-converted. We will use a fresh variable in signals-clauses of the effective specification. Figure 7 shows the normal control flow path through the specification control graph of Figure 6, after α-conversion.

If a given path is traversed in a program execution, then it must be the case that all the specifications along the path hold. We use this to reason inductively about the path's effective specification.

If control flow enters vertex a_1 then $r_1(\sigma_0)$ holds and the formals in the vertex's signature must be bound to the actual arguments. We use the predicate $\text{bind}(x, y)$ to model the binding of actual arguments, results, or exceptions from vertex x to parameters in the signature of vertex y. This binding is according to the parameter passing semantics of AspectJ and Java, a full definition of which is beyond the scope of this paper. As examples, here are the values of bind for the path in Figure 7, respecting the α-conversion shown there:

- $\text{bind}(start, a_1)$ = `(p1==this && dx1==dx && dy1==dy)`
- $\text{bind}(a_1, a_2)$ = `(this==p1 && dx2==dx1 && dy2==dy1)`
- $\text{bind}(a_2, end)$ = `(this==this && dx==dx2 && dy==dy2 && \result==\result2)`

If control flow leaves vertex a_1 on edge $\langle \rho, a_1, y, S_1, \sigma_1 \rangle$ then the implementation code corresponding to that vertex must ensure that the set of possibly mutated locations is f_1 and that if $\rho = \nu$ then $e_1(\sigma_1)$ holds else if $\rho = \varepsilon$ then $s_1(\sigma_1)$ holds.

If control flow exits vertex a_i on a normal edge $\langle \nu, a_i, a_{i+1}, S_i, \sigma_i \rangle$, $1 \le i \le q$, then the set of possibly mutated locations is $f_1 \cup \ldots \cup f_i$ and the following predicate holds:

$$\text{bind}(start, a_1) \wedge \ldots \wedge \text{bind}(a_{i-1}, a_i) \wedge \\ r_1(\sigma_0) \wedge \ldots \wedge r_i(\sigma_{i-1}) \wedge e_1(\sigma_1) \wedge \ldots \wedge e_i(\sigma_i)$$

If control flow exits vertex a_i on an exceptional edge, then $i = q$, and the following predicate holds:

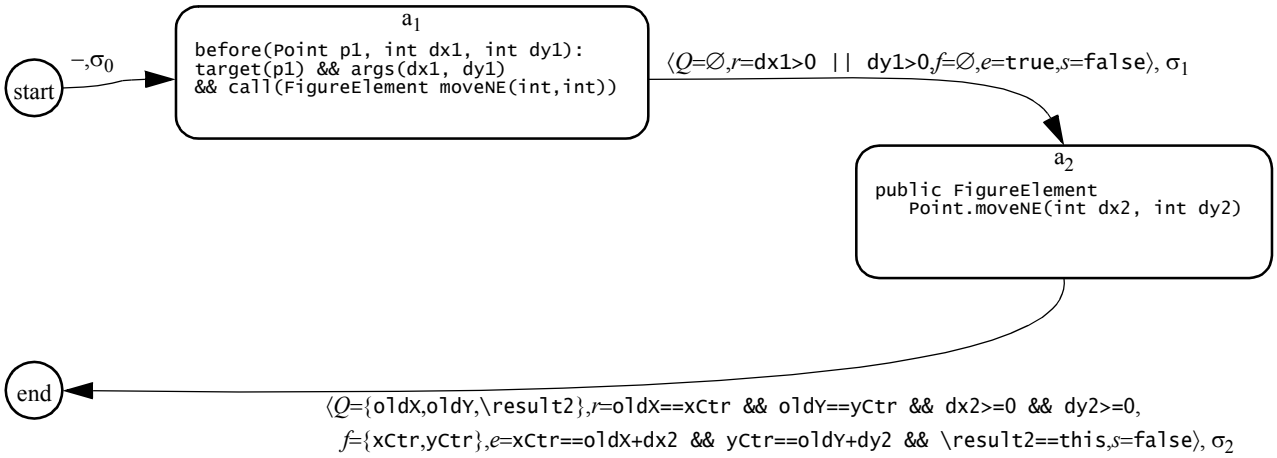

Figure 7: The normal control flow path through the specification composition graph of Figure 6, after α-conversion

$$\text{bind}(start, a_1) \wedge \ldots \wedge \text{bind}(a_{q-1}, a_q) \wedge$$
$$r_1(\sigma_0) \wedge \ldots \wedge r_q(\sigma_{q\text{-}1}) \wedge e_1(\sigma_1) \wedge \ldots \wedge e_{q\text{-}1}(\sigma_{q\text{-}1}) \wedge s_q(\sigma_q)$$

This predicate involves most of the normal postconditions; these just record what happens along the path before the last edge, which is the only one that throws an exception.

To reason about the effective specification from the client's perspective, we must eliminate the intermediate states from these predicates. One way to do this would be to quantify over the states, like:

$$(\forall \sigma_1, \ldots, \sigma_{i-1} \bullet \text{bind}(start, a_1) \wedge \ldots \wedge \text{bind}(a_{q-1}, a_q) \wedge$$
$$r_1(\sigma_0) \wedge \ldots \wedge r_q(\sigma_{q\text{-}1}) \wedge e_1(\sigma_1) \wedge \ldots \wedge e_{q\text{-}1}(\sigma_{q\text{-}1}) \wedge s_q(\sigma_q))$$

However, in JML entire states are not directly expressible, so this idea has to be used indirectly by quantifying over intermediate values of each of the variables used in the predicates. Figure 8 gives the general form of the effective specification along any path. The first line of this general form is calculated by this indirect quantification over intermediate values. Let $\vec{y}$ stand for all the free variables[6] (i.e., field names) in specification cases on a path, and let $\vec{T}$ be their corresponding types. We subscript the names in $\vec{y}$ to represent the value of each named variable in the corresponding state. For example, $\texttt{yCtr}_2$ is a variable whose value is that of the field `yCtr` in state σ_2. Similarly, we write $\vec{y}_i$ to represent the vector of all *i*-subscripted variables, i.e., the vector of values in state σ_i of all variables named in $\vec{y}$.

The second line of Figure 8 gives the explicitly quantified variables of the original specification cases, along with the α-converted parameters from the advice and method signatures.

The requires-, ensures-, and signals-clauses are based on the predicates derived above, with appropriate substitution for the intermediate values of the variables. That is, we write $r_i[\vec{y}:=\vec{y}_{i\text{-}1}]$ for the precondition r_i where for each variable $y \in \vec{y}$, each free occurrence of y is changed to $y_{i\text{-}1}$. We use the same kind of abbreviation for $e_i[\vec{y}:=\vec{y}_i]$.

The general form of the effective specification also must equate the prestate to σ_0 and the poststate to σ_q and include the information provided by the frame axioms. We write $\text{equal}(\vec{y}_i,\vec{y}_j)$ for the predicate that asserts the equality of the variables in $\vec{y}_i$ and $\vec{y}_j$, using == or `.equals` as appropriate for their types, and $\text{notmod}(f,\vec{y},i,j)$ for the predicate that says that variables not listed in the frame f are unchanged; this is defined conceptually as follows (although this quantification is not expressible directly in JML, we can write the equivalent set of conjunctions in any particular case).

$$\text{notmod}(f,\vec{y},i,j) = (\forall y \in \vec{y} \bullet y \notin f \Rightarrow \text{equal}(y_i,y_j))$$

Taken together, we arrive at a single specification case for a single path through the specification composition graph, as Figure 8 shows. Since each possible parallel path is represented by such a specification case, we simply conjoin (using JML's `also` operator) the effective specifications for each path to form the effective specification of the entire invocation (the "parallel composition" alluded to earlier).

Finding the Effective Specification

We can use this formal model to find the effective specification of `Point`'s `moveNE` method with the `PointMoveChecking` assistant.

Consider the path shown in Figure 7. On this path, the free variables are `xCtr` and `yCtr`. Counting the initial state, we need to quantify over 3 states. The effective specification is as shown in Figure 9. Lines from Figure 8 to Figure 9 relate the terms in the general form to the terms in the example. This example specification can be simplified to the following by using transitivity of equality (within clauses), the rule that `false` is the zero of conjunction, and dropping vacuous quantifiers:

```
forall int oldx, oldy;
requires dx >= 0 && dy <= 100
   && oldx == xCtr && oldy == yCtr;
assignable xCtr, yCtr;
ensures xCtr == oldx + dx
   && yCtr == oldy + dy
   && \result == this;
signals (Exception z) false;
```

This is exactly the body of the first specification case arrived at intuitively in Section 1.3. We can analyze the other paths in the graph to calculate the other specification cases. Combining them with `also` yields the full effective specification.

2.1.2. Composition with Around-Advice

Figure 10 gives another assistant, called `PointMoveFixing`. It uses around-advice to change `moveNE` to accommodate negative arguments. Around-advice in AspectJ can execute both before and after the execution of the advised method's body. Unlike before-advice, around-advice can also skip the execution of the advised method's body without throwing an exception. The code (as opposed to the specification) in the body of the advice in Figure 10 illustrates these ideas. If `dx` and `dy` are both non-negative then the statement

6. Though not shown in this paper, JML provides constructs for locally binding names in expressions, such as quantifiers.

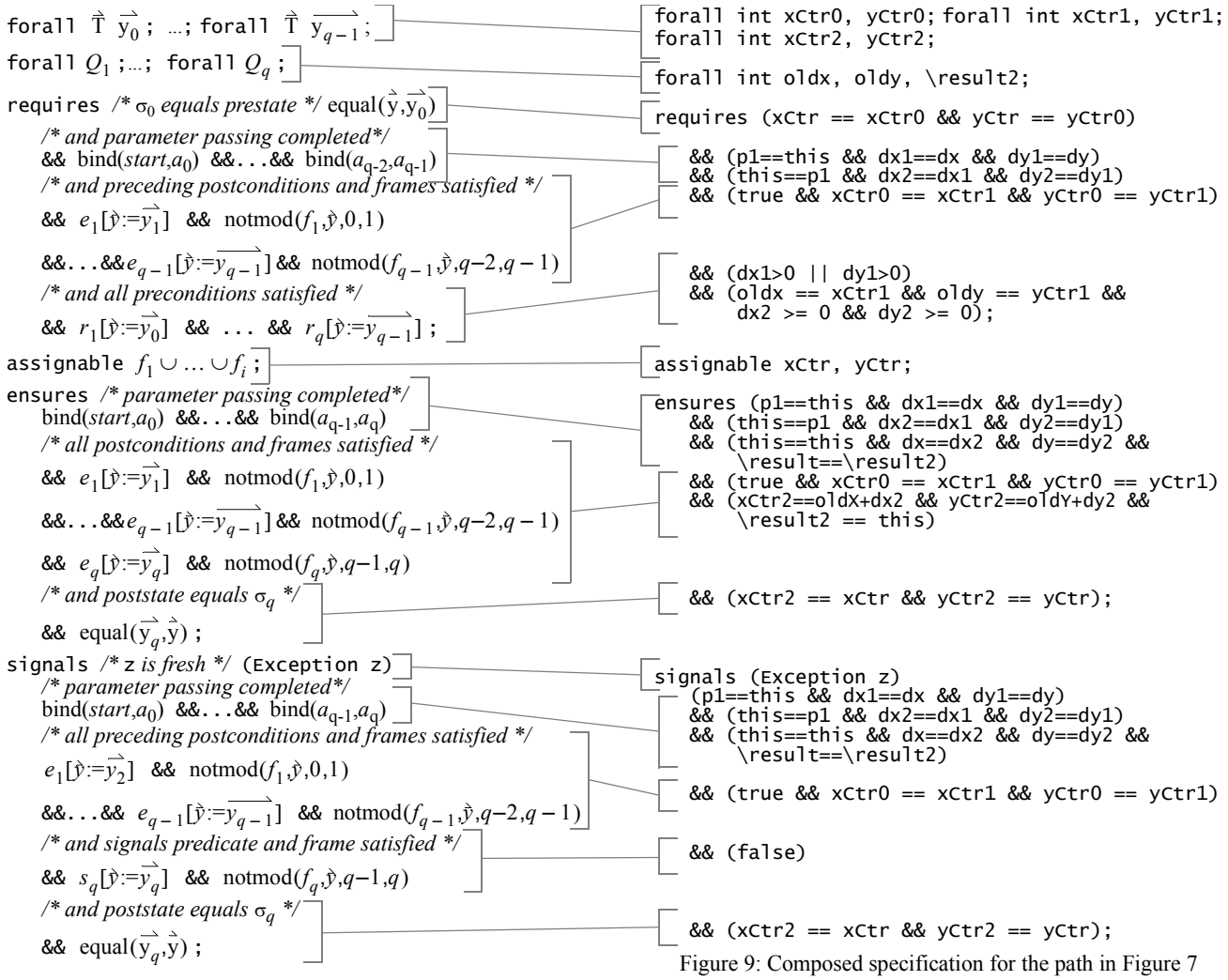

Figure 8: General form of the composed specification for a path

Figure 9: Composed specification for the path in Figure 7
Lines between this and Figure 8 show the correspondence of terms.

```
proceed(p,dx,dy);
```

causes control flow to pass to the original `moveNE` method body with the same arguments as the original invocation. Otherwise, in the else-clause the advice calls the `setX` and `setY` methods on `Point` directly, avoiding the `IllegalArgException` that would be thrown if execution continued into `moveNE`. After the if-statement an acknowledgment message is printed to `System.err`.

Figure 10 also includes a JML specification of the around-advice. As with methods, before-advice, and after-advice, the specification of around advice consists of one or more specification cases joined with the keyword `also`. To specify the additional control flow possible via `proceed` in around-advice, we propose adding the AspectJ proceed-clause to JML as a mechanism for forming compound specification cases. In a specification the proceed-clause joins a specification case called the *before-part*, and a specification case called the *after-part*. The before-part specifies the code executed before proceeding to the original method (and any additional advice if present). The after-part specifies the code executed after returning from the original method (and advice).

The first specification case in Figure 10 (from the beginning up to the `also`) is such a compound specification consisting of before- and after-parts. The case is applicable when `dx` and `dy` are both non-negative, as specified by the requires-clause. In general an ensures-clause in the before-part says that if control flow proceeds to the original method body then the assistant must ensure that the clause's predicate holds. In the example, `ensures true` indicates that control flow can always proceed in this manner. The proceed-clause itself specifies (possibly abstractly) the arguments that will be passed to the original method. A requires-clause in the after-part gives a predicate that can be assumed by the implementation of the after-part. The remainder of the after-part has the usual semantics.

The second specification case, following the `also` keyword in Figure 10, is applicable when at least one of the arguments is negative. The absence of a proceed-clause in this specification case says that control never proceeds to the original method body when this case's precondition is met. The assignable- and ensures-clauses say that control returns to the original client with possible mutation to `p`'s `xCtr` and `yCtr` model fields and the system error stream, and with the given postcondition predicate satisfied.

To reason about effective specifications in the presence of around-advice we would need to extend our formal model. The extension would handle the additional control flow information provided by the proceed-clause. We envision encoding the specification of around-advice with multiple vertices in the specification composition graph. For each piece of around-advice one common vertex would represent the before-parts of all the cases. Separate vertices, one for each case, would represent the after-parts. The edge creation algorithm would require extensions to connect the vertices

```
package foal02;
aspect PointMoveFixing {
   /*@  public behavior
     @    requires dx >= 0 && dy >= 0;
     @    ensures true;
     @   proceed(p,dx,dy);
     @    requires true;
     @    assignable System.err.value;
     @    ensures true;
     @    signals (Exception z) false;
     @ also
     @  public behavior
     @    forall int oldx, oldy;
     @    requires (dx < 0 || dy < 0)
     @             && oldx == xCtr && oldy == yCtr;
     @    assignable p.xCtr, p.yCtr,
     @                System.err.value;
     @    ensures p.xCtr == oldx + dx
     @      && p.yCtr == oldy+ dy;
     @    signals (Exception z) false;
     @*/
   FigureElement around(Point p, int dx, int dy):
      target(p) && args(dx, dy)
      && call(FigureElement moveNE(int,int))
   {
      if (dx >= 0 && dy >= 0) {
         proceed(p,dx,dy);
      } else {
         p.setX( p.getX() + dx );
         p.setY( p.getY() + dy );
      }
      System.err.println("OK");
   }
}
```

Figure 10: An AspectJ module giving around-advice to `Point`

appropriately. The calculation of the composed specification for a given path in the graph would have to account for the expressions in proceed-clauses of the specification.

2.1.3. Summary

We have argued that modular reasoning in aspect-oriented programming languages can be achieved for assistants if we require modules to explicitly accept assistance. We have given a formal model for advice composition that allows us to determine the effective specification of a method. This model also illustrates the reasoning a programmer must undertake even in the absence of formal specifications.

But what impact does our requirement that assistants be explicitly accepted have on the expressiveness of the language? On the one hand, assistants are very expressive in that they are given free rein to change the effective specifications of modules that they assist. On the other hand, requiring assistance to be explicitly accepted dramatically curtails the applicability of assistants. To wit, a common example of the use of aspects is to add tracing capability to an existing program. In a language that just supported explicitly accepted assistance, a programmer would need to make an invasive change to the source code of every module containing a method to be traced (or alternatively, every module calling a method to be traced). We would have gained support for modular reasoning at the expense of modular editing.

2.2 Observers

To resolve this situation we propose that an aspect-oriented programming language should also support a category of aspects that we call observers. An *observer* is an aspect that does not change the effective specification of any other module. Equivalently, an observer may only mutate the state that it owns (in the sense of alias control systems like [20, 21]). It also seems reasonable to allow observers to change accessible global state as well, since a Java module cannot rely on that state not changing during an invocation

```
package foal02;
observer aspect PointMoveTracing {
   private StringBuffer myBuffer =
                              new StringBuffer();
   before(Point p, int dx, int dy):
      target(p) && args(dx, dy)
         && call(FigureElement moveNE(int,int))
   {
      String message = "Entering Point.moveNE" +
         "(" + dx + "," + dy + ")" + "for " + p;
      myBuffer.append(message);
      System.err.println(message);
   }
}
```

Figure 11: An AspectJ module for tracing method calls.

(modulo synchronization mechanisms). The term "observer" is intended to connote the hands-off role of these aspects. We use the term *observation* to discuss the "advice" in an observer.

For example, Figure 11 gives an observer called `PointMoveTracing`. The `observer` modifier declares that this aspect must not change the effective specification of any other module. This observer mutates its own state by appending to `myBuffer` and mutates the global state by printing to `System.err`. However, it does not change the effective pre- or postconditions of `Point`'s `moveNE` method. `PointMoveTracing` merely observes the arguments to the `moveNE` method and reports them. The arguments are passed on to the method unchanged and the method's results are unchanged.

In addition to cross-cutting concerns like tracing, it seems that observers should be useful for logging and as the observer in the observer design pattern [8] (pp. 293–303).

Because observers do not change the effective specifications of the methods they observe, code outside an existing program can apply an observer to any join point in the original program without loss of modular reasoning. In reasoning about the client and implementation code for a method a maintainer of the original program does not need any information from the observer.

2.2.1. Verifying Observerness

The primary challenge of implementing this part of our proposal lies in determining whether a given aspect is really an observer. We envision a static analysis that conservatively verifies this. This analysis is closely related to the problem of verifying frame axioms. In fact we can think of observers as having an implicit frame axiom that prevents modification of locations that are relevant to the receiver and arguments of the observed method.

The main difficulty with statically verifying this lack of relevant side effects is how to deal with aliasing. For example, suppose we have a logging observer that uses an array to track the elements added to some `Set` object. Suppose `Set` uses an array for its representation. If the observer's array and the `Set`'s array are aliased, we might end up with an element being added to the array twice—possibly violating `Set`'s invariant and changing its effective specification. There is a substantial body of work on alias control that may be useful in attacking this [20, 21].

3. EVALUATION

This section briefly evaluates the practical consequences of our proposal. Because we have not yet had the opportunity to develop applications using our proposed restrictions, our evaluation is limited to a review of existing programs. We first consider the aspect-oriented programming guidelines suggested in the ATLAS case study [10]. Then we survey the example aspects from the AspectJ Programmers Guide [2].

3.1 ATLAS Case Study

In the ATLAS case study [10], the authors proposed several guidelines to make working with aspects easier. These were proposed since they had discovered that (p. 346):

> "[t]he extra flexibility provided by aspects is not always an advantage. If too much functionality is introduced from an aspect it may be difficult for the next developer—or the same developer a few months later—to read through and understand the code base."

One of Kersten and Murphy's suggestions is to limit coupling between aspects and classes to promote reuse. Specifically, they suggest that one should avoid the case where an aspect explicitly references a class and that class explicitly references the aspect, since then the class and aspect are mutually dependent. Such mutual dependencies prevent independent reuse. Is this suggestion problematic for our requirement that modules explicitly accept assistance? No, because the suggestion is concerned with mutual dependence between aspects and classes. Suppose an implementation module, *M*, accepts assistance from an assistant, *A*, and *A* changes *M*'s effective specification. This says nothing about whether *M* and *A* are mutually dependent. If *A* explicitly references *M* the modules are mutually dependent. However, if *A* only applies to *M* because of wildcard-based pattern matching and does not explicitly reference *M*, then the modules are not mutually dependent. Next, suppose a client module, *C*, accepts assistance from an assistant, A', and A' only changes the effective specification of modules referenced by *C*, but does not change *C*'s effective specification. In this case A' and *C* are not mutually dependent. In sum, programmers can reduce mutual dependency by having clients accept assistance or by limiting explicit references to classes from assistants.

Kersten and Murphy also suggest using aspects as *factories* by having them provide only after-returning advice on constructors. This after-returning advice mutates the state of every object instantiated to change its default behavior. Limiting the aspects in this way restricts the scope of object–aspect interaction. In our proposal a simple assistant can fill the role of a factory aspect.

For aspects that do not act as factories Kersten and Murphy propose three style rules that restrict the use of aspects (pp. 349–350):

> "Rule #1: Exceptions introduced by a weave must be handled in the code comprising the weave. ... Rule #2: Advise weaves must maintain the pre- and post-conditions of a method. ... Rule #3: Before advise weaves must not include a return statement."

These rules are essentially equivalent to our definition of observers in that they prevent aspects from changing the effective specification of the advised method. Though we propose elevating these style rules to the level of statically checked restrictions.

3.2 Dynamic Aspects

The ATLAS case study uses *dynamic aspects*, or the substitution of different aspect code at runtime to modify the behavior of a program. One way to support this technique within the framework of our proposal would be to have modules accept assistance from abstract assistants. Specifications would be associated with these abstract assistants. The various desired behaviors would be implemented as separate assistants, each extending the abstract assistant and implementing its specification. This approach permits modular reasoning. The language would also need a mechanism to support the runtime selection of a particular concrete assistant.

Related to this idea of dynamic aspects is that of a mechanism for combining observers and other modules. In the current version of AspectJ aspects and classes are combined by naming their modules on the command line in a single invocation of ajc, Xerox's AspectJ compiler. Thus combination takes place "outside the language". To support observation of separately compiled programs, we would like to have a mechanism in the language for instantiating observers. It seems that the same language mechanism might support instantiating observers and selecting concrete assistants.

3.3 Impact of Restrictions

We would like to better understand how our restrictions might limit the practical expressiveness of AspectJ. For a preliminary evaluation we use the examples in the AspectJ Programming Guide to see if our restrictions prohibit any recommended idioms. The programming guide's examples can serve this purpose since they "not only show the features [of AspectJ] being used, but also try to illustrate recommended practice" [2] (from Preface). We separate the example aspects into categories based on how we would implement them with our restrictions. An appendix lists the examples by category; we describe the categories here.

Observers. Many of the example aspects clearly meet our definition of observer. To satisfy our restrictions these would only require the new `observer` annotation.

Assistants. Aspects in the examples that could be implemented as assistants can be divided into two kinds. *Client utilities* are used by client modules to change the effective behavior of objects from other modules. The changes in effective behavior do not affect the representation of those objects. To satisfy our restrictions their assistance would have to be explicitly accepted by the clients. In fact, some of the client utility assistants are declared as nested aspects, i.e., aspects declared inside class declarations. These are similar in spirit to explicitly excepted assistance.

There is one example that could be implemented as an assistant but that is not a client utility. This example uses an aspect to separate a simple concern that cross-cuts a single implementation module. The pointcut, or named join point, for this aspect is declared in the implementation class and the aspect explicitly references the implementation class and the pointcut. To satisfy our restrictions the implementation module would have to explicitly accept the assistance, which would create a mutual dependency. However, this example can be considered a bad design since the concern only cross-cuts the one implementation module. This design flaw can be fixed by nesting the assistant in the implementation module, which would also avoid the mutual dependency.

Dynamic Aspects. To satisfy our restrictions some example aspects would require the dynamic aspect mechanisms alluded to in Section 3.2. One such example is a debugging aspect. This aspect would be an observer, except that it provides after-advice to a GUI frame's constructor to add debugging options to the frame's menu bar. To support this pattern with our restrictions requires the mechanisms for dynamic aspects. The GUI frame would have to accept assistance from an abstract assistant, say `AdditionalMenuConcern`, that allowed a concrete assistant, instantiated at runtime, to add to its menu bar. The debugging aspect would become a concrete assistant extending `AdditionalMenuConcern`. The GUI frame could then be instantiated with the debugging assistant or with an assistant that did nothing.

4. DISCUSSION

As presented, our formal model for reasoning about explicitly accepted assistance does not accommodate advice that applies to join points other than those for method invocations. It seems a simple matter to extend the model to accommodate some other kinds of join points, such as those for field access or exception handling. However, it is not clear whether our model can accommodate dynamic context join points [2], like `cflow(`*pointcut*`)`, which rely on runtime information for applicability tests. It seems that advice

on dynamic context join points can only be modularly reasoned about if this advice is confined to observations. There is one aspect in the AspectJ examples we studied, the `Registry.RegistrationProtection` aspect of the spacewar example, that uses a dynamic context join point with advice that changes the effective specification of the advised method. This example is not supported by the current work.

Because of the generality of aspects without our restrictions and limitations of the target Java Virtual Machine (or *JVM*) [17], AspectJ currently requires whole-program compilation [12]. In our proposal, because assistance is explicitly accepted, it is a simple matter to support separate compilation for modules that accept assistance; the compiler just weaves it into the accepting modules.

On the other hand, observers present interesting challenges for separate compilation. On the surface, since observers do not change the effective specifications of other modules, it should be possible to separately compile them. And indeed this is true—except for the issue of dispatching to observers. The generality of observers means that they can potentially be dispatched to from any join point.

Thus, the only obstacle to separate compilation of AspectJ programs given our restrictions is that of dispatch to observers. Others have suggested that separate compilation for AspectJ is possible using techniques such as specialized class loaders or modified virtual machines [12] (p. 343). With our proposed restrictions the scope of the problem is reduced, likely making it easier to implement these solutions.

5. CONCLUSIONS

To summarize, we have shown that with a few simple modifications AspectJ can support modular reasoning. Our proposal separates aspects into two categories, assistants and observers, which provide complementary features. Assistants are extremely powerful, but require subtle reasoning techniques and are limited in their applicability to maintain modular reasoning. Observers are less powerful but are easy to reason about and are broadly applicable. This broad applicability is achieved by placing heavier burdens on the type system.

A preliminary evaluation showed that for many cases our modifications to the language provide sufficient flexibility. However, we also noted that there is a need for some mechanism to support dynamic aspects.

The other major open problem for our proposal is statically checking that aspects declared as observers meet our definition, as discussed in Section 2.2.1. To attack this problem we propose:

- developing an aspect-oriented calculus for investigating these ideas in a formal setting, and
- developing and proving sound a type-system for the calculus that statically enforces our proposed restrictions on observers.

Other future work on the problem of modular reasoning for aspect-oriented programming languages includes:

- refining our proposed specification constructs for AspectJ and formalizing their semantics, perhaps using something like the refinement calculus [3], and
- investigating behavioral subtyping and formal techniques for verification of aspect-oriented programs.

We are also interested in demonstrating the utility and effectiveness of our ideas by:

- programming non-trivial systems using our restrictions,
- integrating the proposed restrictions into AspectJ, and
- understanding the potential benefits of our restrictions for separate compilation, static analysis, and optimization.

In this paper we have focused on adding support for modular reasoning to the AspectJ language. Future work will also investigate the relevance of our proposal to other aspect-orientation programming languages and techniques, such as composition filters [4], adaptive methods [16], and multidimensional separation of concerns as embodied by Hyper-J [22, 25].

ACKNOWLEDGMENTS

We would like to thank Yoonsik Cheon, Todd Millstein, Markus Lumpe, and Robyn Lutz, for their helpful comments on a draft of this paper. The work of Leavens was supported in part by the US National Science Foundation grants CCR-0097907 and CCR-0113181. The work of both authors was supported in part by a grant from Electronics and Telecommunications Research Institute (ETRI) of South Korea.

APPENDIX

Table 1 below lists the aspects from the examples directory of the Version 1.0.1 release of AspectJ[7]. The second column of the table gives the categorization of each example based on the categories of Section 3.3.

Table 1: Example Aspects and their Categories

Example	Category
telecom/TimerLog	observer
tjp/GetInfo	observer
tracing/lib/AbstractTrace	observer
tracing/lib/TraceMyClasses	observer
tracing/version1/TraceMyClasses	observer
tracing/version2/Trace	observer
tracing/version2/TraceMyClasses	observer
tracing/version3/Trace	observer
tracing/version3/TraceMyClasses	observer
bean/BoundPoint	client utility
introduction/CloneablePoint	client utility
introduction/ComparablePoint	client utility
introduction/HashablePoint	client utility
observer/SubjectObserverProtocol	client utility
observer/SubjectObserverProtocolImpl	client utility
spacewar/Display.DisplayAspect	client utility
spacewar/Display1.SpaceObjectPainting	client utility
spacewar/Display2.SpaceObjectPainting	client utility
telecom/Billing	client utility
telecom/Timing	client utility
spacewar/EnsureShipIsAlive	assistant[a]
coordination/Coordinator	dynamic
spacewar/Debug	dynamic
spacewar/GameSynchronization	dynamic

7. Available from `http://www.aspectj.org`.

Table 1: Example Aspects and their Categories

Example	Category
spacewar/RegistrySynchronization	dynamic
spacewar/Registry.RegistrationProtection	unsupported[b]

a.The EnsureShipIsAlive aspect considered to be a poor design in the discussion of Section 3.3.
b.The aspect, Registry.RegistrationProtection, uses dynamic context join points, which aren't supported by the current work.

REFERENCES

[1] K. Arnold, J. Gosling, and D. Holmes. *The Java Programming Language Third Edition*. Addison-Wesley, Reading, MA, third edition, 2000.

[2] AspectJ Team, the. The AspectJ programming guide. Available from http://aspectj.org/doc/dist/progguide/index.html, Feb. 2002.

[3] R.-J. Back and J. von Wright. *Refinement Calculus: A Systematic Introduction*. Springer-Verlag, 1998.

[4] L. Bergmans and M. Aksits. Composing crosscutting concerns using composition filters. *Commun. ACM*, 44(10):51–57, Oct. 2001.

[5] C. Clifton. MultiJava: Design, implementation, and evaluation of a Java-compatible language supporting modular open classes and symmetric multiple dispatch. Technical Report 01-10, Department of Computer Science, Iowa State University, Ames, Iowa, 50011, Nov. 2001. Available from www.multijava.org.

[6] C. Clifton, G. T. Leavens, C. Chambers, and T. Millstein. MultiJava: Modular open classes and symmetric multiple dispatch for Java. In *OOPSLA 2000 Conference on Object-Oriented Programming, Systems, Languages, and Applications, Minneapolis, Minnesota*, volume 35(10) of *ACM SIGPLAN Notices*, pages 130–145, Oct. 2000.

[7] K. K. Dhara and G. T. Leavens. Forcing behavioral subtyping through specification inheritance. In *Proceedings of the 18th International Conference on Software Engineering, Berlin, Germany*, pages 258–267. IEEE Computer Society Press, Mar. 1996. A corrected version is Iowa State University, Dept. of Computer Science TR #95-20c.

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

[9] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification Second Edition*. The Java Series. Addison-Wesley, Boston, Mass., 2000.

[10] M. A. Kersten and G. C. Murphy. Atlas: A case-study in building a web-based learning environment using aspect-oriented programming. In *Proceedings of the 1999 ACM Conference on Object-Oriented Programming Languages, Systems, and Applications (OOPSLA '99)*, volume 34(10) of *ACM SIGPLAN Notices*, pages 340–352, Denver, CO, November 1999. ACM.

[11] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. Getting started with AspectJ. *Commun. ACM*, 44(10):59–65, Oct. 2001.

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

[13] G. T. Leavens, A. L. Baker, and C. Ruby. JML: A notation for detailed design. In H. Kilov, B. Rumpe, and I. Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 175–188. Kluwer Academic Publishers, Boston, 1999.

[14] G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06p, Iowa State University, Department of Computer Science, Aug. 2001. See verb|www.cs.iastate.edu/ leavens/JML.html|.

[15] K. R. M. Leino. *Toward Reliable Modular Programs*. PhD thesis, California Institute of Technology, 1995. Available as Technical Report Caltech-CS-TR-95-03.

[16] K. Lieberherr, D. Orleans, and J. Ovlinger. Aspect-oriented programming with adaptive methods. *Commun. ACM*, 44(10):39–41, Oct. 2001.

[17] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley Publishing Co., Reading, MA, second edition, 2000.

[18] B. Liskov and J. Wing. A behavioral notion of subtyping. *ACM Trans. Prog. Lang. Syst.*, 16(6):1811–1841, Nov. 1994.

[19] B. Meyer. *Eiffel: The Language*. Object-Oriented Series. Prentice Hall, New York, NY, 1992.

[20] P. Müller. *Modular Specification and Verification of Object-Oriented programs*, volume 2262 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002. The author's PhD Thesis. Available from http://www.informatik.fernuni-hagen.de/import/pi5/publications.html.

[21] J. Noble, J. Vitek, and J. Potter. Flexible alias protection. In E. Jul, editor, *ECOOP '98 – Object-Oriented Programming, 12th European Conference, Brussels, Belgium*, volume 1445 of *Lecture Notes in Computer Science*, pages 158–185. Springer-Verlag, July 1998.

[22] H. Ossher and P. Tarr. Using multidimensional separation of concerns to (re)shape evolving software. *Commun. ACM*, 44(10):43–50, Oct. 2001.

[23] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, Dec. 1972.

[24] A. D. Raghavan and G. T. Leavens. Desugaring JML method specifications. Technical Report 00-03c, Iowa State University, Department of Computer Science, Aug. 2001.

[25] P. L. Tarr, H. Ossher, W. H. Harrison, and S. M. S. Jr. *N* degrees of separation: Multi-dimensional separation of concerns. In *International Conference on Software Engineering*, pages 107–119, 1999.

Source-Code Instrumentation and Quantification of Events

Robert E. Filman
RIACS
NASA Ames Research Center, MS 269/2
Moffett Field, CA 94035 U.S.A.
+1 650–604–1250
rfilman@mail.arc.nasa.gov

Klaus Havelund
Kestrel Technology
NASA Ames Research Center, MS 269/2
Moffett Field, CA 94035 U.S.A
+1 650–604–3366
havelund@email.arc.nasa.gov

ABSTRACT
Aspect-Oriented Programming is making quantified programmatic assertions over programs that otherwise are not annotated to receive these assertions. Varieties of AOP systems are characterized by which quantified assertions they allow, what they permit in the actions of the assertions (including how the actions interact with the base code), and what mechanisms they use to achieve the overall effect. Here, we argue that all quantification is over dynamic events, and describe our preliminary work in developing a system that maps dynamic events to transformations over source code. We discuss possible applications of this system, particularly with respect to debugging concurrent systems.

Categories and Subject Descriptors
D.3.3 [Programming Languages]: Language Constructs and Features – aspects. D.3.2 [Programming Languages] Language Classifications – aspect-oriented programming. D.2.3 [Software Engineering] Coding Tools and Techniques. D.2.5 [Testing and Debugging] Debugging aids.

General Terms
Languages.

Keywords
Quantification, events, dynamic events, debugging, program transformation, model checking.

1. INTRODUCTION
Elsewhere, we have argued that the programmatic essence of Aspect-Oriented Programming is making quantified programmatic assertions over programs that otherwise are not annotated to receive these assertions [10,12]. That is, in an AOP system, one wants to be able to say things of the form, "In this program, when the following happens, execute the following behavior," without having to go around marking the places where the desired behavior is to happen. Varieties of AOP systems are characterized by which quantified assertions they allow, what they permit in the actions of the assertions (including how the actions interact with the base code), and what mechanisms they use to achieve the overall effect. In this paper, we describe our preliminary work in developing a system that takes the notion of AOP as quantification to its logical extreme. Our goal is to develop a system where behavior can be attached to any event during program execution. We describe the planned implementation of this system and discuss possible applications of this technology, particularly with respect to debugging and validating concurrent systems.

2. EVENTS
Quantification implies matching a predicate about a program. Such a predicate must be over some domain. In the quantification/implicit invocation papers, we distinguished between static and dynamic quantification.

Static quantification worked over the structure of the program. That is, with static quantification, one could reference the programming language structures in a system. Examples of such structures include reference to program variables, calls to subprograms, loops, and conditional tests.

Many common AOP implementation techniques can be understood in terms of quantified program manipulation on the static structure of a program. For example, wrapping (e.g., as seen in Composition Filters [1], OIF [11], or AspectJ [19,20]) is effectively embedding particular function bodies in more complex behavior. AspectJ and OIF also provide a call-side wrapping, which can be understood as surrounding the calling site with the additional behavior. An operation such as asserting that class **A**'s use of *x* is the same as class **B**'s use of *y* in Hyper/J [22] can be realized by substituting a reference to a common generated variable for *x* in the text of **A**, and *y* in **B**.

Dynamic quantification, as described in those papers, speaks to matching against events that happen in the course of program execution. An example of dynamic quantification is the jumping-aspect problem [2], where a method behaves differently depending upon whether or not it has been called from within (in the calling-stack sense) a specified routine. Other examples of interesting dynamic events include the stack exceeding a particular size, the fifth unsuccessful call to the login routine with a different password, a change in the number of references to an object, a confluence of variable values (e.g., when $x + y > z$), the blocking of a thread on a synchronization lock, or even a change in the executing thread. The cflow operator in AspectJ is a dynamic quantification predicate.

We are coming to the belief that all events are dynamic. Static quantification should be understood as just the subspecies of events that can be simply inferred, on a one-to-one basis, from the structures of a program. Static quantification is attractive for its straightforward AOP implementation, lower complexity, and independence of programming environment implementation, but unless one starts processing the program comments, there's little

Table 1: Events and event loci

Event	Syntactic locus
Accessing the value of a variable or field	References to that variable
Modifying the value of a variable or field	Assignments to that variable
Invoking a subprogram	Subprogram calls
Cycling through a loop	Loop statements
Branching on a conditional	The conditional statement
Initializing an instance	The constructors for that object
Throwing an exception	Throw statements
Catching an exception	Catch statements
Waiting on a lock	Wait and synchronize statements
Resuming after a lock wait	Other's notify and end of synchronizations
Testing a predicate on several fields	Every modification of any of those fields
Changing a value on the path to another	Control and data flow analysis over statements (slices)
Swapping the running thread	Not reliably accessible, but atomization may be possible
Being below on the stack	Subprogram calls
Freeing storage	Not reliably accessible, but can try using built-in primitives
Throwing an error	Not reliably accessible; could happen anywhere

in the static structure of a program that isn't marked by its dynamic execution.

If the abstract syntax tree is the domain of static quantification, what is the domain of dynamic quantification? Considering the examples in this section, it really has to be events that change the state (both data state and "program counter") of the base language's abstract interpreter. However, defining anything in terms of the abstract interpreter is problematic. First, as was illustrated in Smith's work on 3-Lisp [5], programming languages are not defined in terms of their abstract interpreters. The same language can be implemented with many different interpreters. The set of events generated by one implementation of a language may not correspond to the events generated by another. For example, a run-time environment that manages its own threads is not at all the same as one that relies on the underlying operating system for thread management. Neither is the same as one that takes advantage of the multiple processors of a real multi-processor machine. Second, compilers have traditionally been allowed to optimize—rearrange programs while preserving their input-output semantics. An optimizing compiler may rearrange or elide an "obvious" sequence of expected events. And finally, the data state of the abstract interpreter (including, as it does, all of memory) can be a grand and awkward thing to manipulate.

3. A LANGUAGE OF EVENTS

We view these limitations as bumps in the road, rather than barriers. While we may not be able to capture everything that goes on in a particular interpretive environment, we can get close enough for most practical purposes. The strategy we adopt is to argue that most dynamic events, while not necessarily local to a particular spot in the source code, are nevertheless tied to places in the source code. Table 1 illustrates some primitive events and their associated code loci.

Users are likely to want to express more than just primitive events. The language of events will also want to describe relationships among events, such as that one event occurred before another, that a set of events match some particular predicate, that an event occurred within a particular timeframe, or that no event matching a particular predicate occurred. This suggests that the event language will need (1) abstract temporal relationships, such as "before" and "after," (2) abstract temporal quantifiers, such as "always" and "never", (3) concrete temporal relationships referring to clock time, (4) cardinality relationships on the number times some event has occurred, and (5) aggregation relationships for describing sets of events.

4. SYSTEM ARCHITECTURE

We envision a mechanism where a description of a set of event-action pairs, along with a program, would be presented to a compiler. Each event action pair would include a sentence describing the interesting event in the event language and an action to be executed when that event is realized. Said actions would be programs, and would be parameterized with respect to the elements of the matching events. Examples of such assertions are:

- On every call to method *foo* in a class that implements the interface **B**, replace the second parameter of the call to *foo* with the result of applying method *f* to that parameter.
- Whenever the value of *x+y* in any object of class **A** ever exceeds *5*, print a message to the log and reset *x* to *0*.
- If a call to method *foo* occurs within (some level down on the stack) method *baz* but without an intervening call to method *mumble*, omit the call to method *gorp* in the body of *foo*.

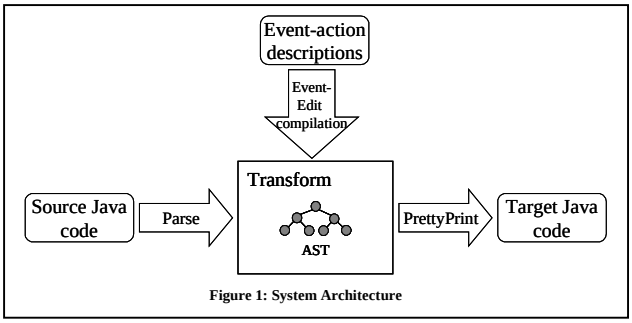

Figure 1: System Architecture

These examples are in natural language. Of course, any actual system will employ something formal.

Clearly, a sufficiently "meta" interpretation mechanism would give us access to many interesting events in the interpreter, enabling a more direct implementation of these ideas. It has often been observed that meta-interpretative and reflective systems can be used to build AOP systems [29]. However, meta-interpreters have traditionally exhibited poor performance. We are looking for implementation strategies where the cost of event recognition is only paid when event recognition is used. This suggests a compiler that would transform programs on the basis of event-action assertions. Such a compiler would work with an extended abstract syntax tree representation of a program. It would map each predicate of the event language into the program locations that could affect the semantics of that event. Such a mapping requires not only abstract syntax tree generation (parsing) and symbol resolution, but also developing primitives with respect to the control and data flow of the program, determining the visibility and lifetimes of symbols, and analyzing the atomicity of actions with respect to multiple threads.

Java compiles into an intermediate form (Java byte codes). In dealing with Java, there is also the choice as to whether to process with respect to the source code or the byte code. Each has its advantages and disadvantages. Byte codes are more real: many of the issues of interest (actual access to variables, even the power consumption of instructions) are revealed precisely at the byte code level. Working with byte codes allows one to modify classes for which one hasn't the source code, including the Java language packages themselves. (JOIE [3] and Jmangler [21] are examples of an AOP systems that perform transformations at the byte code level.) On the other hand, source code is more naturally understandable, allows writing transformations at the human level, and eliminates the need for understanding the JVM and the actions of the compiler. (De Volder's Prolog-based meta-programming system is an example of source-level transformation for AOP [6,7].) We find the complexity arguments appealing. Thus, our implementation plan is to work at the source code level.

5. EXAMPLES

Event quantification is a general framework for supporting aspect oriented programming. It can be used for functionality enhancement, where a program is extended with aspects that add new functionality. For example, a program could be made more reliable by transforming its database update events to also send messages to a backup log. Although functionality extension is a general goal for AOP, we instead discuss some examples within the area of program verification. (In some cases, we expect to be able to extend program behavior for *functionality insurance*: recovering from some classes of program failure.)

In previous work, we studied various program verification techniques for analyzing the correctness of programs. Our work can be classified into two categories: *program monitoring* [17] and *program scheduling* [16,27]. The latter is often called *model checking*.

5.1 Monitoring

Specification-based monitoring consists of monitoring the execution of a program, represented by a sequence of events, by validating the events against a requirements specification. The specification is written in some formal language, typically a temporal logic [24]. For example, a typical requirement is, "Whenever TEMP becomes 100 then within 3 seconds ALARM becomes true." A typical requirement specification has many such assertions. We want to be able to run the program and monitor that specification assertions hold throughout the event trace. The Java PathExplorer system [17] implements this kind of capability. It uses the byte-code engineering tool Jtrek [18] to instrument Java byte code to emit events to an observer, which contains a data structure representing the formulae to be checked. Every event emitted from the running program causes a modification of the data structure. A warning is raised when a specification is violated. We plan to experiment using event quantification at the source code level instead of at the byte code level. The events to be caught are obviously those implicitly referred to in the formula—in the above example, updates to the variables TEMP and ALARM. That is, whenever one of these variables is updated, an event consisting of the variable name, the value, and a timestamp can be emitted to the observer. (The evaluation of the temporal formula can even be performed as part of the quantification action instead of in a separate observer, if real-time performance is not an issue.) Operating on the source code level simplifies creating the instrumentation, as one can work in a high-level language, not byte code. The commercial-available Temporal Rover system performs specification-based monitoring, but does not do automated code instrumentation [8].

Algorithm-based monitoring, like specification-based monitoring, watches the execution of a program emitting events. Rather than matching against user-defined specifications, algorithm-based monitoring uses certain general algorithms for detecting particular kinds of error conditions. Examples are algorithms for detection of deadlock and data race potentials in concurrent programs. These algorithms are interesting since the actual deadlocks or data races do not have to occur in an execution trace in order to be identified as a potential problem. An arbitrary execution trace will normally suffice to identify problems. For example, a cyclic relationship between the locks in a program (thread **T1** takes lock *A* and then *B*, while thread **T2** takes *B* and then *A*) is a potential deadlock. A similar algorithm exists for data races [25]. These algorithms have been implemented in PathExplorer using byte code engineering, and we anticipate trying them out using event quantification.

5.2 Scheduling

Thread scheduling consists of influencing a program's scheduling in order to explore more thread interleavings than would otherwise be achieved with normal testing techniques. As an example, the above mentioned deadlock situation can be explicitly

demonstrated by scheduling the threads such that **T1** takes *A*, and then **T2** immediately takes *B*. Such a schedule might never be seen during normal test of the program. Thread scheduling can be achieved by introducing a centralized scheduler and forcing all threads to communicate with that scheduler when shared data structures (such as locks) are accessed. The scheduler then decides which thread to run, while at the same time keeping track of its scheduling choices. This information can then be used to direct the program to explore new interleavings. We have earlier developed the Java PathFinder system [16, 27] for performing such scheduling analysis using model checking. In order to avoid exploring the reachable subtree below a given program state several times, states are stored in cache, and search is aborted when a state has been visited before. Using quantification, we plan to experiment with state-less model checking [15, 24] where a program's different interleavings are explored, but without storing states. An example of program modification to detect synchronization faults is ConTest [8].

6. RELATED WORK

De Volder and his co-workers [6,7] have argued for doing AOP by program transformation, using a Prolog-based system working on the text of Java programs. We want to extend those ideas to program semantics, combining both the textual locus of dynamic events and transformations requiring complex analysis of the source code.

At the 1998 ECOOP AOP workshop, Fradet and Südholt [13] argued that certain classes of aspects could be expressed as static program transformations. They expanded this argument at the 1999 ECOOP AOP workshop to one of checking for robustness—non-localized, dynamic properties of a system's state [14]. Colcombet and Fradet realized an implementation of these ideas in [4], applying both syntactic and semantic transformations to enforce desired properties on programs. In that system, the user can specify a desired property of a program as a regular expression on syntactically identified points in the program, and the program is transformed into one that raises an exception when the property is violated. Other transformational systems include, *Ku* a notational attempt at formalizing transformation [27], and Schonger et al's proposal to express abstract syntax trees in XML and use XML transformation tools for tree manipulation [26].

Nelson et al. identify three concern-level foundational composition operators: correspondence, behavioral semantics and binding [22]. Correspondence involves identifying names in different entities that are "the same"—for data items, things that should share storage; for functions, functional fragments that need to be assembled into a whole. Behavioral semantics describe how the functional fragments are assembled. Binding is the usual issue of the statics and dynamics of system construction and change. They discuss alternative formal techniques for establishing properties of composed systems within this basis.

Walker and Murphy argue for events as appropriate "join points" for AOP, and that the events exposed by AspectJ are inadequate [32].

7. CONCLUDING REMARKS

In this paper, we've examined the idea of implementing AOP systems as programs transformed by quantified responses to dynamic events. Two comments about the place of such a system in the order of things are worth making:

- We've been talking about implementation environments, not software engineering. An underlying implementation does not imply anything about the "right" organization of "separate concerns" to present to a user. In particular, we have been completely agnostic about the appropriate structure for the actions of action-event pairs. It may be the case that unqualified use of an event language with raw action code snippets is a software engineering wonder, but we doubt it.

- An environment that can map from quantified dynamic events to modified code would be an excellent environment for experimenting with and building systems for AOP. In some sense, these ideas can be viewed as a domain-specific language for developing aspect-oriented languages.

8. ACKNOWLEDGMENTS

Our thanks to Tarang Patel and Tom Pressburger for their comments on the draft of this paper.

9. REFERENCES

[1] Bergmans, L., and Aksit, M. Composing crosscutting concerns using composition filters. *Comm. ACM Vol. 44*, No. 10, 2001, pp. 51–57.

[2] Brichau, J., De Meuter, W., and De Volder, K. Jumping aspects. Workshop on Aspects and Dimensions of Concerns, ECOOP 2000, Cannes, France, Jun. 2000. http://trese.cs.utwente.nl/Workshops/adc2000/papers/Brichau.pdf

[3] Cohen, G. Recombining concerns: Experience with transformation. First Workshop on Multi-Dimensional Separation of Concerns in Object-oriented Systems (OOPSLA '99), Oct. 1999, www.cs.ubc.ca/~murphy/multid-workshop-oopsla99/position-papers/ws23-cohen.pdf

[4] Colcombet, T. and Fradet, P. Enforcing trace properties by program transformation. *Proc. 27th ACM Symp. Principles of Programming Languages*, Boston, Jan. 2000, pp. 54–66.

[5] des Rivieres, J. and Smith, B. C. The implementation of procedurally reflective languages. *Conf. Record of the 1984 ACM Symposium on LISP and Functional Programming*, Austin, Texas, Aug. 1984, pp. 331–347.

[6] De Volder, K., Brichau, J., Mens, K., and D'Hondt, T. Logic meta-programming, a framework for domain-specific aspect programming languages. http://www.cs.ubc.ca/~kdvolder/binaries/cacm-aop-paper.pdf

[7] De Volder, K., and D'Hondt, T. Aspect-oriented logic meta programming. *Proceedings of Meta-Level Architectures and Reflection, Second International Conference, Reflection'99. LNCS 1616*, Springer-Verlag, 1999, pp. 250–272.

[8] Drusinsky, D. The Temporal Rover and the ATG Rover. *SPIN Model Checking and Software Verification, LNCS* 1885, K. Havelund, J. Penix, and W. Visser (Eds.) Springer, 2000, pp. 323–330.

[9] Edelstein, O., Farchi, E., Nir, Y., Ratsaby, G., Ur, S. Multithreaded Java program test generation. *IBM Systems Journal, Vol. 41*, No. 1, 2002, pp. 111–125.

[10] Filman, R.E. What is aspect-oriented programming, revisited. Workshop on Advanced Separation of Concerns, 15th European Conference on Object-Oriented Programming, Budapest, Jun. 2001. http://trese.cs.utwente.nl/Workshops/ecoop01asoc/papers/Filman.pdf

[11] Filman, R. E., Barrett, S., Lee, D. D., and Linden, T. Inserting ilities by controlling communications. *Comm. ACM, Vol. 45,* No. 1, Jan. 2002, pp. 116–122.

[12] Filman, R. E. and Friedman, D. P. Aspect-oriented programming is quantification and obliviousness. Workshop on Advanced Separation of Concerns, OOPSLA 2000, Minneapolis, Oct. 2000. http://trese.cs.utwente.nl/Workshops/OOPSLA2000/papers/filman.pdf

[13] Fradet, P. and Südholt, M. Towards a generic framework for aspect-oriented programming, Third AOP Workshop, *ECOOP'98 Workshop Reader, LNCS, 1543*, pp. 394–397, Jul. 1998. trese.cs.utwente.nl/aop-ecoop98/papers/ Fradet.pdf

[14] Fradet, P and Südholt, M. An aspect language for robust programming. Int. Workshop on Aspect-Oriented Programming, ECOOP, Jun. 1999. http://trese.cs.utwente.nl/aop-ecoop99/papers/fradet.pdf

[15] Godefroid P. Model checking for programming languages using VeriSoft. *Proc. of 24th ACM Symp. on Principles of Programming Languages*, Paris, Jan. 1997, pp. 174–186.

[16] Havelund K. and Pressburger T. Model checking Java programs using Java PathFinder. *International Journal on Software Tools for Technology Transfer, Vol. 2*, No. 4, Apr. 2000, pp. 366–381.

[17] Havelund K. and Rosu, G. Monitoring Java programs with Java PathExplorer. In *Proceedings of the First International Workshop on Runtime Verification (RV'01), Electronic Notes in Theoretical Computer Science, Vol. 55*, No. 2, Elsevier Science, Paris, Jul. 2001.

[18] Jtrek. Compaq. http://www.compaq.com/java/download/jtrek

[19] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., and Griswold, W. G. An overview of AspectJ, *Proceedings ECOOP 2001*, J. L. Knudsen (Ed.) Berlin: Springer-Verlag LNCS 2072, pp. 327–353.

[20] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., and Griswold, W. G. Getting started with AspectJ. *Comm. ACM Vol. 44*, No. 10, 2001, pp. 59–65.

[21] Kniesel, G., Costanza, P., and Austermann, M. JMangler–A Framework for Load-Time Transformation of Java Class Files. Proc. First IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2001*)*, Florence, Nov. 2001, http://www.informatik.uni-bonn.de/~costanza/SCAM_jmangler.pdf

[22] Nelson, T., Cowan, D. and Alencar, P. Supporting formal verification of crosscutting concerns. *Metalevel Architectures and Separation of Crosscutting Concerns: Third International Conference, Reflection 2001*, A. Yonezawa and S. Matsuoka (Eds.) Sep. 2001, Kyoto, Berlin: Springer-Verlag, LNCS 2192, pp. 153-169.

[23] Ossher, H. and Tarr, P. The shape of things to come: Using multi-dimensional separation of concerns with Hyper/J to (re)shape evolving software. *Comm. ACM Vol. 44*, No. 10, 2001, pp. 43–50.

[24] Pnueli A. The temporal logic of programs. *Proc. 18th IEEE Symp. Foundations of Computer Science*, 1977, pp. 46–57.

[25] Savage S., Burrows M., Nelson G., Sobalvarro P., and Anderson T. Eraser: A dynamic data race detector for multi-threaded programs. *ACM Transactions on Computer Systems, Vol. 15*, No. 4, Nov. 1997.

[26] Schonger, S., Pulvermueller, E., and Sarstedt, S. Aspect oriented programming and component weaving: using XML representations of abstract syntax trees. Workshop Aspektorientierte Softwareentwicklung, Institut für Informatik III, Universität Bonn, Feb. 2002, i44w3.info.uni-karlsruhe.de /~pulvermu/workshops/aosd2002/submissions/schonger.pdf.

[27] Skipper, M. A Model of composition oriented programming, Proc. Workshop on Multi-Dimensional Separation of Concerns in Software Engineering, Int'l Conf/ on Software Engineering, Limerick, Ireland, June 2000, www.research.ibm.com/hyperspace/workshops/icse2000/Papers/skipper.pdf

[28] Stoller S. D. Model-checking multi-threaded distributed Java programs. *International Journal on Software Tools for Technology Transfer*, in press.

[29] Sullivan, G. T. Aspect-oriented programming using reflection and meta-object protocols. *Comm. ACM Vol. 44*, No. 10, 2001, pp. 95–97.

[30] Teitelman, W. and Masinter, L. The Interlisp programming environment. *Computer Vol. 14*, No. 4, 1981, pp. 25–34.

[31] Visser W., Havelund K., Brat G., and Park S. Model checking programs. *Proc. ASE'2000: The 15th IEEE Intl. Conf. Automated Software Engineering*, Sep. 2000, pp. 3–12.

[32] Walker, R. J. and Murphy, G. C. Joinpoints as ordered events: towards applying implicit context to aspect-orientation. Workshop on Advanced Separation of Concerns in Software Engineering at ICSE, Toronto, May, 2001, www.research.ibm.com/hyperspace/workshops/icse2001/Papers/walker.pdf.