Formalizing the Temporal Order of Join Points *

Paul C. Attie Sergei Kojarski David H. Lorenz
Northeastern University
College of Computer & Information Science
Boston, Massachusetts 02115 USA

E-mail: {attie,kojarski,lorenz}@ccs.neu.edu

Abstract

Crucial to the programming and debugging of complex aspect-oriented programs is the ability to specify and monitor temporal ordering amongst join-points. We present a temporal logic specification of such constraints in AspectJ. This provides a formal semantics for join-point occurrences and is a step towards the formal verification of AOP programs. We illustrate the usefulness of this approach through the formalization and implementation of aspects in AspectJ.

1 Introduction

One purpose of Aspect-Oriented Programming (AOP) is to make software clearer by modularizing cross-cutting concerns into aspects. However, in the current state of AOP, an unfortunate consequence of using aspects is the difficulty of predicting what the aspect-oriented program does, even for small programs. This difficulty is especially apparent when debugging and testing programs in AspectJ [17, 12].

AspectJ is easy to learn up to a certain point, but is difficult to master. Even experts do not have a complete knowledge of AspectJ's semantics, and cannot predict the behavior of programs with confidence. Often programs are developed in a trial-and-error fashion. Because of this difficulty, the full potential of AOP has not been realized.

One reason for the difficulty of writing and debugging programs in AspectJ is the lack of explicit knowledge of the temporal order of join points. Furthermore, there is currently no complete documentation for AspectJ of the order that join points may occur in an execution.

A good formal model for join point semantics will be a great help to AOSD designers and programmers; it will enable them to write more correct, and sophisticated code. It will also provide a foundation for verification methods for AOP.

In this paper we identify this shortcoming and illustrate the problem concretely through code examples. We propose a method for addressing the problem with a formal model in temporal logic. We use AspectJ and temporal logic for the illustration, but the essential idea can be carried out using other languages and formalizms.

In the next section we show code examples where the result is unexpected without a formal model. In Section 3 we describe an essential part of a formal model. In Section 4 we return to the examples of Section 2 and illustrate how the model is helpful in predicting their wrong behavior and synthesizing a correct aspect for the problem.

2 Motivation

Aspect-Oriented Programming in AspectJ is done by means of advising join points. Simple aspects, like logging the occurrence of certain join points, involve a single join point at a time. Generally, however, an aspect may involve several kinds of join points, and rely on the temporal order in which these join points occur within a program execution. The precise temporal order is significant for the correctness of the aspects.

For some kinds of join points, e.g., for a method-call and a method-execution, the temporal order is common knowledge. A method-call is (normally[1]) followed by a method-execution of the method. Advice to method-execution may rely on this fact and safely access the enclosing method-call join point via the **`thisEnclosingJoinPointStaticPart`** construct (for an example see [16]). However, for most other combinations of join points, the precise dependency is not well doc-

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

[1] Unless diverted with an **`around`** advice.

umented and not well understood, and consequently is the source of confusion, incorrect programs, and frustration.

Consider an aspect that monitors the join points in an execution of a program in order to keep a record of the inheritance relationship between classes of instantiated objects.[2] Such an aspect may be useful when reflection is not available or when the available reflective information is not complete [18]. Elsewhere [13, 15, 14], we describe a whole system of aspects, *Aspectual Reflection*, which collectively provide an alternative to Java Core Reflection [4] in AspectJ.

The (simplified) `Hierarchy` class and the `Monitor` aspect are part of that Aspectual Reflection system. `Hierarchy` is a global repository of class-class associations. `Monitor` collects subclass-superclass pairs and stores them in `Hierarchy`. `Hierarchy` provides access to the stored information via a **`static public`** `getSuperClass` method:

```
public class Hierarchy {
  static public Class getSuperClass(Class
      subclass) {
    return (Class)map.get(subclass);
  }
  static void put(Class subclass, Class
      superclass) {
    if (superclass==null) return;
    map.put(subclass, superclass);
  }
  static java.util.HashMap map =
    new java.util.HashMap();
}
```

The `Monitor` aspect defines two abstract pointcuts, consHappens and initHappens, which monitor construction of objects and initialization of classes. When construction occurs, the order in which initialization occurrences follow is analyzed. Relying on the fact that superclasses must be initialized before their subclasses, `Monitor` reconstructs the inheritance relationship.

```
abstract aspect Monitor {
  protected pointcut consHappens():
    call(program..*.new(..));
  abstract pointcut initHappens();
  before(): initHappens() {
    Class subclass = thisJoinPoint.
      getSignature().getDeclaringType();
    Hierarchy.put(subclass,superclass);
    superclass=subclass;
  }
  protected Class superclass = null;
}
```

For this scheme to work, all that is left to do is to complete the defenition of the two pointcuts consHappens and initHappens in a concrete subaspect [11]. In the next sections, we describe three attempts to define these pointcuts, illustrating the process of evolving an aspect-oriented implementation in AspectJ and the difficulty of evolving a correct aspect.

[2] Ignoring classes with no objects.

2.1 First Attempt

A sensible implementation strategy seems to be one that monitors constructor-call and static-initialization join points.

> **Rationale 1:** Classes are loaded to the JVM the first time they are instantiated, and super-classes are always loaded before subclasses.

When a class is instantiated, its constructor is run, triggering a constructor-call join point. When a class is loaded, its static initializer is run, triggering a static-initialization join point. For example, suppose that `Derived` extends `Base`,

```
package program; //test program 1
class Base {}
class Derived extends Base {}
```

Instantiating `Derived` would trigger the following join point sequence:

```
<constructor-call Derived>
 <static-initialization Base>
 </static-initialization Base>
 <static-initialization Derived>
 </static-initialization Derived>
</constructor-call Derived>
```

Therefore, the sequence of static-initialization join points that occur within a constructor-call reflects the class hierarchy. `Attempt1` extends `Monitor`, resets `superclass` on constructor-call, and records an association on static-initialization.

```
aspect Attempt1 extends Monitor {
  before(): consHappens() {
    superclass = null;
  }
  pointcut initHappens():
    staticinitialization(program..*);
}
```

`Attempt1` will correctly record, for the class hierarchy of test porgram 1, the inheritance association:

- `Derived` $\leq$ `Base` (correct)

However, for a slightly different program, `Aspect1` fails. Suppose that `Base` included a static initialization code, in which an `Outsider` class is instantiated:

```
package program; //test program 2
class Base {static {new Outsider();}}
class Derived extends Base {}
class Outsider {}
```

When `Derived` is instantiated for test program 2, the static initializer of `Base` triggers a constructor-call join point for the `Outsider` class:

```
<constructor-call Derived>
 <static-initialization Base>
  <constructor-call Outsider>
   <static-initialization Outsider>
   </static-initialization Outsider>
  </constructor-call Outsider>
 </static-initialization Base>
 <static-initialization Derived>
 </static-initialization Derived>
</constructor-call Derived>
```

`Attempt1` then records an incorrect inheritance fact:

- `Derived` $\leq$ `Outsider` (incorrect)

and will not record the correct relationship:

- `Derived` $\leq$ `Base` (missing)

The problem with `Attempt1` is that a constructor-call may be nested in the control flow of a static-initialization.

2.2 A Second Attempt

An alternative strategy is to monitor static-initialization join points, and to use the aspect instance specification **`percflow`**.

> **Rationale 2:** Only immediate static-initialization children of a constructor-call are relevant.

The corresponding `Attempt2` aspect,

```
aspect Attempt2 extends Monitor
   percflow(consHappens()) {
  pointcut initHappens():
    staticinitialization(program..*);
}
```

works well for the two test programs. Advising test program 2 captures the class hierarchy:

- `Derived` $\leq$ `Base` (correct)

because the join points would be:

```
<constructor-call Derived>
 <static-initialization Base>
  <constructor-call Outsider>
   <static-initialization Outsider>
   </static-initialization Outsider>
  </constructor-call Outsider>
 </static-initialization Base>
 <static-initialization Derived>
 </static-initialization Derived>
</constructor-call Derived>
```

However, `Attempt2` is not correct either: it will not work correctly on the following test program 3:

```
package program; //test program 3
class Base {}
class Derived extends Base {}
class Derived2 extends Base {}
```

For two consequent instantiations of `Derived` and `Derived2`,

```
   Derived d = new Derived();
   Derived2 d2 = new Derived2();
```

the corresponding constructor calls produce the following join point sequence:

```
<constructor-call Derived>
 <static-initialization Base>
 </static-initialization Base>
 <static-initialization Derived>
 </static-initialization Derived>
</constructor-call Derived>
<constructor-call Derived2>
 <static-initialization Derived2>
 </static-initialization Derived2>
</constructor-call Derived2>
```

Consequently, `Attempt2` will record an incomplete inheritance relationship. It will record:

- `Derived` $\leq$ `Base` (correct)

but will not record the relationship:

- `Derived2` $\leq$ `Base` (missing)

The problem with `Attempt2` is that each class is loaded only once to the JVM. Hence, after a class is loaded, subsequent instantiations of that class or its subclasses will not contain a static-initialization join point for that class. As a result, subclass and superclass relationships may not be reflected in the static-initialization join point sequences, causing `Attempt2` to generally fail.

2.3 A Third Attempt

A remification of the problem in `Attempt2` would seem to be the replacement of static-initialization with initialization. The first indicates initialization of classes, the latter the initialization of objects and sub-objects:

```
aspect Attempt3 extends Monitor
   percflow(consHappens()) {
  pointcut initHappens():
    initialization(program..*.new(..));
}
```

Rationale 3: Object construction triggers the execution of initializers for the sub-objects according to the class hierarchy, starting from the root. Each initializer execution triggers an initialization join point. Therefore, the sequence of initialization join points correspond to the class hierarchy.

`Attempt3` does not suffer from the deficiency of `Attempt2` because initializers for the sub-objects occur each time a new object is constructed. `Attempt3` constructs a correct class hierarchy for the test programs shown so far. For example, the counterexample for `Attempt2` (test program 3) produces the join point sequence:

```
<constructor-call Derived>
 <initialization Base>
 </initialization Base>
 <initialization Derived>
 </initialization Derived>
</constructor-call Derived>
<constructor-call Derived2>
 <initialization Base>
 </initialization Base>
 <initialization Derived2>
 </initialization Derived2>
</constructor-call Derived2>
```

and `Attempt3` correctly identifies the inheritance relationships:

- `Derived` $\leq$ `Base` (correct)
- `Derived2` $\leq$ `Base` (correct)

Unfortunately, however, `Attempt3` is also incorrect. Consider test program 4:

```
package program; //test program 4
interface I {}
class Derived extends Base implements I {}
```

`new Derived()` results in:

```
<constructor-call Derived>
 <initialization I>
 </initialization I>
 <initialization Base>
 </initialization Base>
 <initialization Derived>
 </initialization Derived>
</constructor-call Derived>
```

that fools the `Attempt3` aspect to record:

- `Base` $\leq$ `I` (incorrect)
- `Derived` $\leq$ `Base` (correct)

Surprisingly enough,[3] `AspectJ` triggers `initialization` join points for interfaces.

2.4 Assessment

The lack of a formal model of the temporal order of join points makes the job of AspectJ programmers extremely difficult. Expert domain knowledge evolves ad hoc by making assumptions about the expected temporal order of join points and revising these assumptions when they fail to satisfy special cases. A formal model would provide a much more rigorous approach.

The motivating examples demonstrate that intuitive assumptions about the temporal order of join points may be misleading and often lead to incorrect code. Moreover, the actual join point sequences in AspectJ often include join points that a Java programer would not have expected, e.g., `initialization` join points for interfaces. When join points are "unexpected" they are not addressed during testing and are a likely source for problems in an actual execution later.

Furthermore, in the absence of a formal model, there is no guarantee that an aspect's code will work correctly for any program. Note, that the aspects above are extremely simple. Yet, even for these simple examples, it is not clear what can go wrong. Even with a rich set of test cases, there is room for serious flaws. What about inner classes, static inner classes, or local classes? In the absence of a formal model it is extremely difficult to prove correctness of aspect code.

3 A Temporal Model for Join Points

We present a model formalizing eight out of the ten kinds of join points in AspectJ: method-call, method-execution, constructor-call, initialization, constructor-execution, static-initialization, field-get, and field-set. The formalization of the handler and object pre-initialization join points is omitted due to space considerations.

3.1 The Temporal Logic CTL

We use CTL for the formalization. CTL is a propositional branching time temporal logic [8]. We have the following syntax for CTL, where p denotes an atomic proposition (boolean variable), and f, g denote (sub-)formulae.

[3] Initializers are not allowed for Java interfaces, and there is no `staticinitialization` join point for interfaces

1. Each of p, $f \wedge g$ and $\neg f$ is a formula (where $\wedge$, $\neg$ indicate conjunction and negation, respectively).
2. $\mathsf{EX}f$ is a formula which means that there is an immediate successor state reachable by executing one step, in which formula f holds.
3. $\mathsf{AX}f$ is a formula which means that, in every successor state reachable by executing one step, formula f holds.
4. $\mathsf{A}[f\,\mathsf{U}\,g]$ is a formula which means that for every computation path, there is some state along the path where g holds, and f holds at every state along the path until that state.
5. $\mathsf{E}[f\,\mathsf{U}\,g]$ is a formula which means that for some computation path, there is some state along the path where g holds, and f holds at every state along the path until that state.
6. $\mathsf{A}[f\,\mathsf{U_w}\,g]$ is a formula which means that for every computation path, f holds at every state along the path until g holds (if ever), and that f holds at all states along the path if g never holds along the path.
7. $\mathsf{E}[f\,\mathsf{U_w}\,g]$ is a formula which means that for some computation path, f holds at every state along the path until g holds (if ever), and that f holds at all states along the path if g never holds along the path.

We define the abbreviations $\mathsf{AF}f$ (eventually) for $\mathsf{A}[true\,\mathsf{U}f]$, $\mathsf{AG}f$ (always) for $\mathsf{A}[f\,\mathsf{U_w}\,false]$. We note that the $\mathsf{EX}f$, $\mathsf{A}[f\,\mathsf{U}\,g]$, and $\mathsf{E}[f\,\mathsf{U}\,g]$ modalities are sufficient to express all the others [8]. We give the others explicitly for presentation purposes; they will be used heavily in the sequel.

3.2 Representing Join Points as Intervals

We represent each event as a proposition with the same name. The proposition is false initially and becomes true when the event is executed. An interval A is an ordered pair of events $(\mathit{start}(A),\ \mathit{finish}(A))$, which give the start and finish of A within a program execution. We also express intervals by naming the start and finish events explicitly, e.g., (S, F). A join point *jp* is an interval (S, F) (Figure 1), where $S = \mathit{start}(jp)$ is the event that triggers AspectJ's "`before()`:*jp*" advice, and the event $F = \mathit{finish}(jp)$ is the event that triggers "`after()`:*jp*" advice. We denote by $\mathcal{I}$ the universal set of intervals.

3.3 Join Point Interface

We assume that each interval is associated with data (environmental information), accesible through a join point interface. We use a simplified version of the join point interface found in AspectJ:

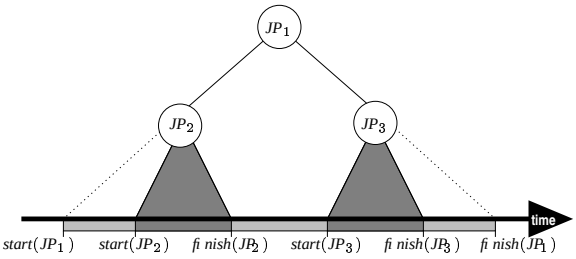

Figure 1. Mapping join point trees to temporal intervals

join point interface

$jp : \langle \mathit{this},\ \mathit{target},\ \mathit{args},\ \mathit{signature} \rangle$
$\mathit{this}, \mathit{target} :\ \mathit{Object}$
$\mathit{args} :\ \mathit{List}(\mathit{Object})$
$\mathit{signature} :\ \langle \mathit{declaringType},\ \mathit{name} \rangle$

We also define a set of accessor functions used to access join point data:

accessor functions

$\mathit{this}(jp) = \pi_1(jp)$
$\mathit{target}(jp) = \pi_2(jp)$
$\mathit{args}(jp) = \pi_3(jp)$
$\mathit{declaringType}(jp) = \pi_1(\pi_4(jp))$
$\mathit{name}(jp) = \pi_2(\pi_4(jp))$

3.4 Interval Types

There are two sorts of intervals: code intervals and access intervals (Figure 2). A code interval is either of: initialization, static-initialization, constructor-execution, method-execution. An access interval is either of: method-call, constructor-call, field-get, and field-set.

code

$\mathsf{code}(jp) \equiv \mathsf{initialization}(jp) \vee \mathsf{static\text{-}initialization}(jp) \vee$
$\mathsf{method\text{-}execution}(jp) \vee \mathsf{constructor\text{-}execution}(jp)$

access

$\mathsf{access}(jp) \equiv \mathsf{field\text{-}set}(jp) \vee \mathsf{field\text{-}get}(jp) \vee$
$\mathsf{method\text{-}call}(jp) \vee \mathsf{constructor\text{-}call}(jp)$

3.5 Operators on Intervals

A join point cannot be undone, i.e., events are stable:

stability

$\mathsf{AG}[e \Rightarrow \mathsf{AG}e]$

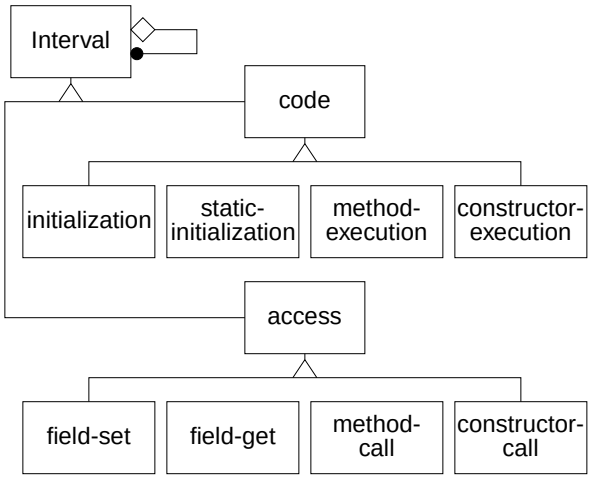

Figure 2. Intervals

For example, once an interval A starts, $start(A)$ remains true forever. We abbreviate by $in(A)$ the fact that the current event occurs within the interval A:

in

$$in(A) \equiv start(A) \wedge \neg\, finish(A)$$

Every two intervals are either disjoint or one is properly contained in the other:

nesting

$$\begin{array}{l}\forall\ A, B \in \mathcal{I} \\ \mathsf{AG}[\neg\ (in(A) \wedge in(B)) \vee \\ \quad [(start(B) \Rightarrow start(A)) \wedge \\ \quad\ (finish(A) \Rightarrow finish(B))] \vee \\ \quad [(start(A) \Rightarrow start(B)) \wedge \\ \quad\ (finish(B) \Rightarrow finish(A))]]\end{array}$$

An interval cannot finish unless it started:

causality

$$\begin{array}{l}\forall\ A \in \mathcal{I} \\ \mathsf{AG}[finish(A) \Rightarrow start(A)]\end{array}$$

We denote by $A\ ;\ B$ that the interval A is followed immediately by the interval B:

binary ;

$$\begin{array}{l}(S_1,\ F_1)\ ;\ (S_2,\ F_2) \equiv \\ \quad S_1 \wedge \mathsf{A}[in(S_1,\ F_1)\ \mathsf{U_w}\ (\mathsf{AX}S_2)] \wedge \\ \quad \mathsf{AG}[F_1 \Rightarrow \mathsf{AF}S_2] \wedge \\ \quad \mathsf{AG}[\neg\ (in(S_1,\ F_1) \wedge in(S_2,\ F_2))] \\ e\ ;\ I \equiv (e,\ e)\ ;\ I \equiv I\ ;\ (e,\ e) \equiv I\ ;\ e\end{array}$$

Similarly, $A_1;\ A_2;\ \ldots;\ A_n$ denotes that A_{i-1} is followed immediately by A_i for $i = 2, 3, \ldots, n$:

n-ary ;

$$\begin{array}{l}(S_1,\ F_1)\ ;\ (S_2,\ F_2)\ ;\ \ \ldots\ ;\ (S_n,\ F_n) \equiv \\ \quad S_1 \wedge \bigwedge_i \mathsf{A}[in(S_i,\ F_i)\ \mathsf{U_w}\ (\mathsf{AX}S_{i+1})] \wedge \\ \quad \bigwedge_i \mathsf{AG}[F_i \Rightarrow \mathsf{AF}S_{i+1}] \\ \quad \bigwedge_{i,j} \mathsf{AG}[\neg\ (in(S_i,\ F_i) \wedge in(S_j,\ F_j))]\end{array}$$

"If p, then the interval occurs" is abbreviated by:

$\twoheadrightarrow$ definition

$$p \twoheadrightarrow (S_1, F_1) = (\neg\, p \vee S_1,\ \neg\, p \vee F_1)$$

3.6 Auxiliary Functions

We also use functions from the Java language domain:

auxiliary functions

Returns true if type is class
$isClass : Type \rightarrow Boolean$
Returns true if class is static
$isStatic : Class \rightarrow Boolean$
Returns true if class is found in JVM
$isLoaded : Class \rightarrow Boolean$
Returns true if class has a superclass
$hasSuperClass : Class \rightarrow Boolean$
Returns superclass
$superclass : Class \rightarrow Class$
Returns true if class implements interface
$implements : Class \rightarrow Interface \rightarrow Boolean$
Returns enclosing class for inner class
$enclosing : Class \rightarrow Class$
Dispatches message and returns result method
$dispatch : message \rightarrow method$
where
$T : Type$
$Type : Class \mid Interface$
$message : \langle receiver,\ name,\ args \rangle$
$method : \langle declaringClass,\ name \rangle$
$receiver : Object$
$args : List(Object)$
$declaringClass, C : Class$

3.7 Constraints on Intervals

Temporal order of intervals is not arbitrary. For example, an access interval contains a code interval, and each code interval may contain only access intervals. To specify these constraints we use CTL. The formulas below illustrate the idea by imposing constraints on the order of access and static-initialization join points:

1. $\mathsf{AG}(start(code(jp)) \Rightarrow$
   $isLoaded(declaringType(jp)))$

2. $\mathsf{AG}\,(start\,(access\,(jp)) \Rightarrow$
   $\mathsf{AF}\,(isLoaded\,(declaringType\,(jp))))$

3. $\mathsf{AG}\,(subclass(c_1, c_2) \wedge isLoaded\,(c_1) \Rightarrow$
   $isLoaded\,(c_2))$

4. $\mathsf{AG}\,(isLoaded\,(c) \Rightarrow$
   $\mathsf{AG}\,(\neg\ (start(\text{static-initialization}(jp))$
   $\vee\ finish(\text{static-initialization}(jp)))$
   $\wedge\ declaringType\,(jp) = C))$

5. $\mathsf{AG}(start(access(jp)) \Rightarrow$
   $\mathsf{AX}(isLoaded(declaringType(jp))$
   $\vee\ LoadClass(declaringType(jp))))$

3.8 Temporal Order of Join Points

Complete specification of temporal order of join points using elementary CTL formulas would be exhaustive. To ease the task, we unify formulas expressing constraints over same-kind intervals in rules. The model presented provides one rule for each join point kind. For example, a constructor-call join point is specified in a ***constructor-call*** rule. Similarities between different join point intervals are captured in a set of helper rules.

3.8.1 Helper Rules

The ***LoadClass*** rule specifies events that occurr during a class load and initialization. ***LoadClass*** also defines how class loads related to one another. For any argument class C not yet loaded into the JVM, a sequence of four intervals occur: (1) for static (inner) classes, the root (first non-static) enclosing class of C is loaded; (2) the superclass of C (if exists) is loaded; (3) static-initialization (join point) for C. (4) The last interval specifies no events but ensures that classes are loaded only once per execution.

LoadClass

$$
\begin{array}{l}
LoadClass(C) \equiv (!isLoaded(C) \twoheadrightarrow \\
\quad [(isStatic(C) \twoheadrightarrow \\
\quad\ LoadClass(staticEncRoot(C)))\ ; \\
\quad (hasSuperClass(C) \twoheadrightarrow \\
\quad\ LoadClass(superclass(C)))\ ; \\
\quad \text{static-initialization}(jp)\ ; \\
\quad [isLoaded := isLoaded' + \langle C, true\rangle]]) \\
where \\
declaringType(jp) = C \\
staticEncRoot(C) \equiv \\
\quad (isStatic(C) \Rightarrow staticEncRoot(\\
\quad\ enclosing(C))) \vee C
\end{array}
$$

The ***Exec*** rule is an abbriviation that specifies execution of a sequence of intervals φ as a sequence of its members' executions.

Exec

$$
\begin{array}{l}
Exec(\varphi) \equiv \\
\quad \bigvee_{(S,\ F)=head(\varphi)} (S,\ F)\ ;\ Exec(tail(\varphi))
\end{array}
$$

The ***CodeEval*** rule defines events that may occur within a code interval as a sequence of arbitrary number of access intervals.

CodeEval

$$
\begin{array}{l}
CodeEval \equiv \\
\quad \bigvee_{\varphi} Exec(\varphi) \\
where \\
\varphi \equiv \text{access}(jp_1), \text{access}(jp_2), \ldots
\end{array}
$$

Instantiation of a class C is defined by the ***Instantiate*** rule. First, the superclass of C (if exists) is instantiated. Next, interfaces implemented by C and not by its superclass are instantiated. The last interval is an initialization join point for C.

Instantiate

$$
\begin{array}{l}
Instantiate(C) \equiv \\
\quad (hasSuperClass(C) \twoheadrightarrow \\
\quad\ Instantiate(superclass(C)))\ ; \\
\quad (\forall I\ ((implements(C, I) \wedge \\
\quad\ \neg\ (hasSuperClass(C) \wedge \\
\quad\ implements(superclass(C), I)) \twoheadrightarrow \\
\quad\ initialization(jp')))\ ; \\
\quad initialization(jp) \\
where \\
declaringType(jp) = C \\
declaringType(jp') = I
\end{array}
$$

3.8.2 Join Point Rules

A join point is an interval restricted by start and finish events of `thisJoinPoint`, and contains other intervals as specified by the following rules.

A constructor-call join point encloses the ***LoadClass*** interval followed by the ***Instantiate*** interval.

constructor-call

$$
\begin{array}{l}
\text{constructor-call}(jp) \equiv \\
\quad start(\text{constructor-call}(jp))\ ; \\
\quad LoadClass(declaringType(jp))\ ; \\
\quad Instantiate(declaringType(jp))\ ; \\
\quad finish(\text{constructor-call}(jp))
\end{array}
$$

Not surprisingly, a method-call contains a method-execution join point. However, if the declaringType for the

method is a class (rather than an interface), then ***LoadClass*** for this class precedes method-execution.

method-call

method-call(jp) $\equiv$
  start(method-call(jp)) ;
  (*isClass*(*declaringType*(jp)) $\twoheadrightarrow$
   LoadClass(*declaringType*(jp))) ;
  method-execution(jp') ;
  finish(method-call(jp))
where
this(jp') = *target*(jp)
args(jp') = *args*(jp)
declaringType(jp') = π_1(*dispatch*(
  ($\langle$*target*(jp), *name*(jp), *args*(jp)$\rangle$)))
name(jp) = *name*(jp')

field-set and field-get join points contain only a ***LoadClass*** interval.

field-set

field-set(jp) $\equiv$
  start(field-set(jp)) ;
  LoadClass(*declaringType*(jp)) ;
  finish(field-set(jp))

field-get

field-get(jp) $\equiv$
  start(field-get(jp)) ;
  LoadClass(*declaringType*(jp)) ;
  finish(field-get(jp))

The initialization, static-initialization, method-execution and constructor-execution join points contain a ***CodeEval*** interval. An initialization join point also includes at least one constructor-execution join point. Moreover, initialization may also include other constructor-execution join points if its declaringType is a class.

initialization

initialization(jp) $\equiv$
  start(initialization(jp)) ;
  CodeEval ;
  constructor-execution(jp') ;
  $\bigvee_{\varphi}$ *Exec*(φ) ;
  finish(initialization(jp))
where
(*isClass*(*declaringType*(jp)) $\wedge$
  $\varphi \equiv$ constructor-execution(jp'')
  $\forall$ constructor-execution(jp'') $\in \varphi$
   (*this*(jp'') = *this*(jp) = *this*(jp')) $\wedge$
   (*declaringType*(jp'') = *declaringType*(jp) =
   declaringType(jp'))) $\vee$
  $\varphi = \emptyset$

static-initialization

static-initialization(jp) $\equiv$
  start(static-initialization(jp)) ;
  CodeEval ;
  finish(static-initialization(jp))

method-execution

method-execution(jp) $\equiv$
  start(method-execution(jp)) ;
  CodeEval ;
  finish(method-execution(jp))

constructor-execution

constructor-execution(jp) $\equiv$
  start(constructor-execution(jp)) ;
  CodeEval ;
  finish(constructor-execution(jp))

4 Applying the Temporal Model to Aspects

The use of temporal logic brings to bear a large corpus of verification methods that have been developed for temporal logic over the last 25 years. In particular *model checking* [7] and deductive verification [19] are the most mature methods. We use deductive verification in this paper, implicitly using a deductive system for CTL [9]. Model checking is also attractive since it is mechanical, and thus requires little manual proof effort. In the longer term, method for synthesizing programs from temporal logic specifications may also be applicable, particularly when concurrency enters the picture [10, 5, 6]

4.1 Verification

The temporal assumptions of an aspect can be verified against the model to ensure their correctness (or detect its incorrectness) for any possible sequence of join points. For example, the aspects `Attempt1`, `Attempt2`, and `Attempt3` presented in Section 2 can be proven faulty directly from the rules in the model.

- To see that `Attempt1` may fail, we can observe that ***static-initialization*** contains ***CodeEval***, which contains an access, which in turn may contain a nested static-initialization join point.
- To see that `Attempt2` may fail, we can observe that in ***LoadClass***, static-initialization is guarded by *isLoaded* which prevents it from being loaded more than once. This contradict the assumption in `Attempt2` that each constructor-call of a class contains static-initialization join points for all its superclasses.

- To see that `Attempt3` may fail, we can observe that ***Instantiate*** includes not only class initialization but also interface initialization join points, which are mistakenly interpreted as classes by the aspect.

4.2 Testing

The model can be used to generate test cases at a desired level of coverage. Consider an aspect that advises constructor-call join points and uses **`thisEnclosingJoinPointStaticPart`** within the advice body. According to the model, the **`thisEnclosingJoinPointStaticPart`** expression may be the static part of any code joint point, and never of an access join point. Therefore, it is important to test such an aspect for all the code join point kinds and there is no need to test for access join points.

4.3 Synthesis

The model can be used to derive "temporally correct" aspects. For example, we can now derive an `Attempt4` aspect that is correct for all cases.

To derive correct code, we start by analyzing the ***constructor-call*** rule, since it is the only join point related to inheritance. As described earlier, ***static-initialization*** join points do not allow us to build a complete hierarchy. A ***static-initialization*** join point occurs only once within program execution, as ensured by the `isLoaded` guard in ***LoadClass*** rule.

The ***Instantiate*** rule includes only two kinds of initialization join points. The first is related to interface initialization and the second is related to class initialization. Class initializations occurs for all classes, starting from the rootclass, going down the inheritance links. Therefore, by inspecting only class-related initialization join points that are enclosed in a constructor-call, we may record the inheritance relationships for the instantiated class and all its superclasses.

Looking at ***constructor-call***, we see that constructor-call may contain other constructor calls, directly or indirectly. Therefore, we need to intercept only initialization join points immediately nested in a constructor-call. This is achieved by adding the **`percflow`** construct to the aspect definition.

We observe from the model, that initialization join points are used only by ***Instantiate***, which appears only within the constructor-call rule. Therefore, join points, available to an aspect defined as **`percflow`**`(consHappens())` will not contain unrelated initialization join points.

The complete code for the correct `Attempt4` aspect, which was derived from the model, is presented below:

```
aspect Attempt4 extends Monitor
   percflow(consHappens()) {
  pointcut initHappens():
    initialization(*.new(..))
    && if(!thisJoinPoint.getSignature().
        getDeclaringType().isInterface());
}
```

4.4 Deriving New Properties

A formal model also allows to derive non-trivial interesting temporal properties of join points, which typically are not known even to AspectJ programmers. The following is a partial list:

- field-get and field-set may have a non-empty **`cflowbelow`**.
- There are initialization and constructor join points for interfaces (as for classes) but no static-initialization join points (unlike for classes).
- A static-initialization join point for a static inner class always occurs after a static-initialization of its enclosing root class (***LoadClass*** helper rule).
- A constructor-execution join point is not a child of the corresponding constructor-call but rather a child of initialization.

5 Conclusion and Related Work

Knowledge of temporal ordering amongst join points is crucial in programming and debugging complex aspect-oriented programs. The main contribution of this paper is in applying temporal logic to formalize the order of join points of a given AOP model. A complete model for the temporal order of join points is:

- a basis for rigorous, unambiguous, and well defined semantics for the join point model;
- a first step toward verification methods for AOP, which can leverage off previous temporal logic work.

Aberg *et al.* evolved an OS kernel using temporal logic and AOP [1, 2]. Their work, however, does not provide a foundation for reasoning about join points temporal order in AspectJ. We have applied temporal logic to formalize the temporal semantics of join points.

We illustrate that a precise model for the temporal order of join points help to verify, test, and synthesis aspect code. We have used this model in our own aspect-oriented software development. Aspectual Reflection [13, 15, 18, 14] heavily relies on the join point temporal order in AspectJ. It would have been close to impossible to implement Aspectual Reflection correctly without the formal model presented in this paper. A formal model would also be useful for verifying backward compatability of new AspectJ versions. Exteding our model to AspectJ 1.1 is future work.

References

[1] R. A. AAberg, J. L. Lawall, M. Südholt, G. Muller, and A.-F. Le Meur. Evolving an OS kernel using temporal logic and aspect-oriented programming. In *Proceedings of the Second AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software*, Boston, MA, Mar. 2003. Published as Northeastern University College of Computer and Information Science Technical Report NU-CCIS-03-03, `http://www.ccs.neu.edu/home/lorenz/aosd2003`.

[2] R. A. AAberg, J. L. Lawall, M. Südholt, G. Muller, and A.-F. Le Meur. On the automatic evolution of an os kernel using temporal logic and aop. In *Proceedings of the 18th IEEE International Conference on Automated Software Engineering (ASE 2003)*, Montreal, Canada, Oct. 2003.

[3] AOSD 2003. *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development*, Boston, Massachusetts, Mar. 17-21 2003. ACM Press.

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

[5] P. C. Attie and E. A. Emerson. Synthesis of concurrent systems with many similar processes. *ACM Transactions on Programming Languages and Systems*, 20(1):51–115, Jan. 1998.

[6] P. C. Attie and D. Lorenz. Establishing behavioral compatibility of software components without state-explosion. Technical Report NU-CCIS-0302, College of Computer and Information Science, Northeastern University, Boston, MA, Mar. 2003. http://www.ccs.neu.edu/homer/lorenz/papers/reports/NU-CCS-03-02.html.

[7] E. M. Clarke, E. A. Emerson, and P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, Apr. 1986. Extended abstract in Proceedings of the 10th Annual ACM Symposium on Principles of Programming Languages.

[8] E. A. Emerson. Temporal and modal logic. In J. V. Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, *Formal Models and Semantics*. The MIT Press/Elsevier, Cambridge, Mass., 1990.

[9] E. A. Emerson. Temporal and modal logic. In J. V. Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, *Formal Models and Semantics*. The MIT Press/Elsevier, Cambridge, Mass., 1990.

[10] E. A. Emerson and E. M. Clarke. Using branching time temporal logic to synthesize synchronization skeletons. *Sci. Comput. Programming*, 2:241 – 266, 1982.

[11] E. Ernst and D. H. Lorenz. Aspects and polymorphism in AspectJ. In AOSD 2003 [3], pages 150–157.

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

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

[14] S. Kojarski and D. H. Lorenz. Reflective mechanisms in AOP languages. Technical Report NU-CCIS-03-07, College of Computer and Information Science, Northeastern University, Boston, MA 02115, Mar. 2003.

[15] S. Kojarski, D. H. Lorenz, and R. Hirschfeld. Aspectual reflection. Submitted for publication, Sept. 2003.

[16] K. Lieberherr, D. H. Lorenz, and P. Wu. A case for statically executable advice: Checking the Law of Demeter with AspectJ. In AOSD 2003 [3], pages 40–49.

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

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

[19] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag, 1992.