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 formalisms.

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) 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 Scientific Software at Northeastern University.

1Unless 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. 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 definition 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.

2.1 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.

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.

```
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>

    Attempt1 will correctly record, for the class hierarchy of test program 1, the inheritance association:
    • Derived ≤ 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>
</constructor-call Outsider>
</static-initialization Base>
</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
percfow(consHappens()) { }
pointcut initHappens():
staticinitialization(program..*);
}

works well for the two test programs. Advising test pro-
gram 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>

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 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 inheri-
tance 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, sub-
sequent 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 re-
lected in the static-initialization join point sequences, caus-
ing 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 initializa-
tion. The first indicates initialization of classes, the latter
the initialization of objects and sub-objects:
```java
aspect Attempt3 extends Monitor
    perflow(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:

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

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:

```java
package program; //test program 4
interface I {}
class Derived extends Base implements I {}
new Derived() results in:
```

```xml
<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 programmer 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.

\(^8\)Initializers are not allowed for Java interfaces, and there is no static-initialization join point for interfaces.
1. Each of \( p, f \land g \) and \( \neg f \) is a formula (where \( \land, \neg \) indicate conjunction and negation, respectively).

2. \( \text{EX} f \) is a formula which means that there is an immediate successor state reachable by executing one step, in which formula \( f \) holds.

3. \( \text{AX} f \) is a formula which means that, in every successor state reachable by executing one step, formula \( f \) holds.

4. \( \text{A}[Ug] \) 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. \( \text{E}[Ug] \) 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. \( \text{A}[Uwg] \) 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. \( \text{E}[Uwg] \) 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 \( \text{AF} f \) (eventually) for \( \text{A}[\text{true} U f] \), \( \text{AG} f \) (always) for \( \text{A}[\text{U} \text{false} U f] \). We note that the \( \text{EX} f, \text{A}[Ug], \) and \( \text{E}[Ug] \) 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 \((\text{start}(A), \text{finish}(A))\), 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 \( S = \text{start}(jp) \) is the event that triggers AspectJ’s “\( \text{before}() : jp \)” advice, and the event \( F = \text{finish}(jp) \) is the event that triggers “\( \text{after}() : jp \)” advice. We denote by \( I \) 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](image)

```plaintext
join point interface
jp : \langle this, target, args, signature\rangle
target \( \text{target} : \text{Object} \)
args : \text{List(Object)}
signature : \langle \text{declaringType}, \text{name}\rangle
```

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

```plaintext
accessor functions
this(jp) = \pi_3(jp)
target(jp) = \pi_2(jp)
args(jp) = \pi_3(jp)
declaringType(jp) = \pi_4(jp))
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.

```plaintext
code,
code(jp) \equiv \text{initialization}(jp) \lor \text{static-initialization}(jp) \lor \text{method-execution}(jp) \lor \text{constructor-execution}(jp)
```

```plaintext
access,
access(jp) \equiv \text{field-set}(jp) \lor \text{field-get}(jp) \lor \text{method-call}(jp) \lor \text{constructor-call}(jp)
```

### 3.5 Operators on Intervals

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

```plaintext
\text{stability}
\text{AG}[e \Rightarrow \text{AG}e]
```
For example, once an interval $A$ starts, $\text{start}(A)$ remains true forever. We abbreviate by $\text{in}(A)$ the fact that the current event occurs within the interval $A$:

$$\text{in}(A) \equiv \text{start}(A) \land \neg \text{finish}(A)$$

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

$$\forall A, B \in \mathcal{I} \quad \mathcal{AG}[\neg (\text{in}(A) \land \text{in}(B)) \lor 
\text{[(start}(B) \Rightarrow \text{start}(A)) \land 
\text{[(finish}(A) \Rightarrow \text{finish}(B)) \land 
\text{[(start}(A) \Rightarrow \text{start}(B)) \land 
\text{[(finish}(B) \Rightarrow \text{finish}(A))]]]$$

An interval cannot finish unless it started:

$$\forall A \in \mathcal{I} \quad \mathcal{AG}[(\neg \text{finish}(A) \Rightarrow \text{start}(A))]$$

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

$$\mathbin{;}$$

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

$$\quad$$

“If $p$, then the interval occurs” is abbreviated by:

$$\rightarrow \mathbf{definition} \quad p \Rightarrow (S_1, F_1) \equiv (\neg p \lor S_1, \neg p \lor F_1)$$

### 3.6 Auxiliary Functions

We also use functions from the Java language domain:

$$\text{auxiliary functions}$$

- $\text{isClass : Type} \rightarrow \text{Boolean}$
- $\text{isStatic : Class} \rightarrow \text{Boolean}$
- $\text{isLoaded : Class} \rightarrow \text{Boolean}$
- $\text{hasSuperClass : Class} \rightarrow \text{Boolean}$
- $\text{superclass : Class} \rightarrow \text{Class}$
- $\text{enclosing : Class} \rightarrow \text{Class}$
- $\text{dispatch : message} \rightarrow \text{method}$

We denote by $\mathcal{AF}$ the set of 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:

### 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. \( \text{AG}(\text{start}(\text{code}(jp))) \Rightarrow \text{isLoaded}(\text{declaringType}(jp))) \)

2. \( \text{AG}(\text{start}(\text{access}(jp))) \Rightarrow \text{AF}(\text{isLoaded}(\text{declaringType}(jp))) \)

3. \( \text{AG}(\text{subclass}(c_1, c_2) \land \text{isLoaded}(c_1) \Rightarrow \text{isLoaded}(c_2)) \)

4. \( \text{AG}(\text{isLoaded}(c) \Rightarrow \text{AG}(\neg(\text{start}(\text{static-initialization}(jp))) \lor \text{finish}(\text{static-initialization}(jp))) \land \text{declaringType}(jp) = C) \)

5. \( \text{AG}(\text{start}(\text{access}(jp))) \Rightarrow \text{AX}(\text{isLoaded}(\text{declaringType}(jp))) \lor \text{LoadClass}(\text{declaringType}(jp))) \)

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 occur during a class load and initialization. LoadClass also defines how class loads relate to one another. For any argument class \( C \) not yet loaded into the JVM, a sequence of four intervals occurs: (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.

\[ \text{LoadClass} \]
\[
\text{LoadClass}(C) \equiv (\text{isLoaded}(C) \Rightarrow \\
(\neg \text{isStatic}(C) \Rightarrow \\
\text{LoadClass}(\text{staticEncRoot}(C))) ; \\
(\neg \text{hasSuperClass}(C) \Rightarrow \\
\text{LoadClass}(\text{superclass}(C))) ; \\
\text{static-initialization}(jp) ; \\
[\text{isLoaded} \equiv \text{isLoaded} + \langle C, \text{true} \rangle]]
\]

\text{where}
\[
\text{declaringType}(jp) = C \\
\text{staticEncRoot}(C) \equiv \\
\text{isStatic}(C) \Rightarrow \text{staticEncRoot}(\text{enclosing}(C)) \lor C
\]

The Exec rule is an abbreviation that specifies execution of a sequence of intervals \( \varphi \) as a sequence of its members’ executions.

\[ \text{Exec} \]
\[
\text{Exec}(\varphi) \equiv \\
\text{V}_{(S, F)=\text{head}(\varphi)} (S, F) ; \text{Exec}(\text{tail}(\varphi))
\]

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

\[ \text{CodeEval} \]
\[
\text{CodeEval} \equiv \\
\text{V}_{\varphi} \text{Exec}(\varphi) \\
\text{where} \\
\varphi \equiv \text{access}(jp_1), \text{access}(jp_2), \ldots
\]

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 \( C \).

\[ \text{Instantiate} \]
\[
\text{Instantiate}(C) \equiv \\
(\neg \text{hasSuperClass}(C) \Rightarrow \\
\text{Instantiate}(\text{superclass}(C))) ; \\
(\forall I (\langle \text{implements}(C, I) \land \\
\neg (\text{hasSuperClass}(C) \land \\
\text{implements}(\text{superclass}(C), I)) \Rightarrow \\
\text{initialization}(jp')))) ; \\
\text{initialization}(jp)
\]

\text{where}
\[
\text{declaringType}(jp) = C \\
\text{declaringType}(jp') = I
\]

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.

\[ \text{constructor-call} \]
\[
\text{constructor-call}(jp) \equiv \\
\text{start}(\text{constructor-call}(jp)) ; \\
\text{LoadClass}(\text{declaringType}(jp)) ; \\
\text{Instantiate}(\text{declaringType}(jp)) ; \\
\text{finish}(\text{constructor-call}(jp))
\]

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 \textit{LoadClass}
for this class precedes method-execution.

\begin{verbatim}
method-call
method-call\((jp)\) ≡
  start(method-call\((jp)\)) ;
  (isClass(declaringType\((jp)\)) →
    LoadClass(declaringType\((jp)\))) ;
  method-execution\((jp')\) ;
  finish(method-call\((jp)\))
where
  this\((jp')\) = target\((jp)\)
  args\((jp')\) = args\((jp)\)
  declaringType\((jp')\) = \(π_1\)(dispatch(
    ((target\((jp)\), name\((jp)\), args\((jp)\)))))
  name\((jp)\) = name\((jp')\)
\end{verbatim}

field-set and field-get join points contain only a \textit{LoadClass} interval.

\begin{verbatim}
field-set
field-set\((jp)\) ≡
  start(field-set\((jp)\)) ;
  LoadClass(declaringType\((jp)\)) ;
  finish(field-set\((jp)\))
\end{verbatim}

\begin{verbatim}
field-get
field-get\((jp)\) ≡
  start(field-get\((jp)\)) ;
  LoadClass(declaringType\((jp)\)) ;
  finish(field-get\((jp)\))
\end{verbatim}

The initialization, static-initialization, method-execution and constructor-execution join points contain a \textit{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.

\begin{verbatim}
initialization
initialization\((jp)\) ≡
  start(initialization\((jp)\)) ;
  CodeEval ;
  constructor-execution\((jp')\) ;
  \(∀\varphi\) Exec(\(\varphi\)) ;
  finish(initialization\((jp)\))
where
  (isClass(declaringType\((jp)\)) ∧
   \(\varphi\) ≡ constructor-execution\((jp'')\) ∧
   ∃ constructor-execution\((jp'')\) ∈ \(\varphi\) ∧
   (this\((jp'')\) = this\((jp)\) = \(this\((jp)\)\)) ∧
   (declaringType\((jp'')\) = declaringType\((jp)\) =
    declaringType\((jp')\)) \(∀\varphi = \emptyset\)
\end{verbatim}

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 contradicts 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 join 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 a 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 occur 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:

```java
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 compatibility of new AspectJ versions. Extending our model to AspectJ 1.1 is future work.
References


