Verification of Aspect-UML models using Alloy

F. Mostefaoui   J. Vachon

DIRO - University of Montreal, Canada

Aspect Oriented Modeling Workshop at AOSD 2007
Issue at hand.

Aspect A: Timing connections before JP

Aspect B: Billing for connections before JP

JP: Drop connections

Telecom application

Aspect interactions
Possible conflict between Timing and Billing.
Issue at hand.

Aspects A and B in a Telecom application:
- **Aspect A**: Timing connections before JP
- **Aspect B**: Billing for connections before JP
- **JP**: Drop connections

Aspect interactions:
Possible conflict between Timing and Billing.
Outline.

1. Introduction
2. Aspect-UML profile
   - A simple example.
   - Static view.
   - Semantic annotations.
3. Overview of Alloy
4. Translation of an Aspect-UML model into an Alloy specification
   - Translation of static elements.
   - Translation of dynamic elements.
5. Verification
6. Summary
INTRODUCTION.

Aspect interactions

- Aspect/Aspect.
- Aspect/Base system.

Aspect interactions can induce:

- Violation of local properties and invariants within aspects.
- Violation of local properties and invariants within the base program.
Aspect interactions

- Aspect/Aspect.
- Aspect/Base system.

Aspect interactions can induce:

- Violation of local properties and invariants within aspects.
- Violation of local properties and invariants within the base program.
Aspects interactions
  - Aspect/Aspect.
  - Aspect/Base system.

Aspect interactions can induce:
  - Violation of local properties and invariants within aspects.
  - Violation of local properties and invariants within the base program.
Aspect interactions

- Aspect/Aspect.
- Aspect/Base system.

Aspect interactions can induce:

- Violation of local properties and invariants within aspects.
- Violation of local properties and invariants within the base program.
INTRODUCTION.

Our objective . . .

- Prevent undesirable interactions implying aspects.
- Work out detection at some abstract level (i.e. independent from programming languages).
  - Use annotations to capture semantic information.

Provide a modeling and verification methodology for AO systems.
INTRODUCTION.

Our objective ...

- Prevent undesirable interactions implying aspects.
- Work out detection at some abstract level (i.e. independent from programming languages).
  - Use annotations to capture semantic information.

Provide a modeling and verification methodology for AO systems.
INTRODUCTION.

Our objective . . .

- Prevent undesirable interactions implying aspects.
- Work out detection at some abstract level (i.e. independent from programming languages).
  - Use annotations to capture semantic information.

Provide a modeling and verification methodology for AO systems.
Our modeling and verification methodology uses:
- Aspect-UML profile to model AO systems.
- Alloy Analyzer for the verification.
INTRODUCTION.

Our methodology process
A simple telephony application from the AspectJ Guide.

- Basic functionalities: initiate, drop and merge connections.
- Additional features: timing and billing for connections.
Aspect-UML Profile.
Class diagram.
Aspect-UML Profile.
Class diagram.

```
Customer
+name : string

Device
+status: string
+phoneNumber: int

Connection
+status : string
+origin: Device
+destination: Device
+complete()
+drop()
+merge()
```
Aspect-UML Profile.
Class diagram.
Aspect-UML Profile.
Class diagram.

- **Customer**
  - `+name : string`

- **Device**
  - `+d_status: string`
  - `+phoneNumber: int`

- **Connection**
  - `+status : string`
  - `+origin: Device`
  - `+destination: Device`
  - `+complete() <-
    +drop() <-
    +merge()`

- **Timer**
  - `+startTime : int`
  - `+connectionTime: int`
  - `+c: Connection`
  - `+getTimer(c: Connection): Timer`
  - `after OpComplete(c: Connection) {}`
  - `after OpDrop(c: Connection) {}`

- **<aspect> Timing**
  - `Bill`
  - `+charge : int`
  - `+client: Customer`
  - `getBill(cust: Customer): Bill`
  - `after OpDrop(c: Connection) {}`
Aspect-UML profile.
Semantic annotations.

Specification of operations.
- Pre and Post conditions specifying the individual contracts to be met by advices and join points.

Context passing constraints.
- Relationship defining how the execution context is passed from join points to related advices.

Precedence constraints.
- Precedence relationship to specify the execution order between aspects.
Aspect-UML profile.

Semantic annotations.

**Specification of operations.**
- Pre and Post conditions specifying the individual contracts to be met by advices and join points.

**Context passing constraints.**
- Relationship defining how the execution context is passed from join points to related advices.

**Precedence constraints.**
- Precedence relationship to specify the execution order between aspects.
Aspect-UML profile.
Semantic annotations.

**Specification of operations.**
- Pre and Post conditions specifying the individual contracts to be met by advices and join points.

**Context passing constraints.**
- Relationship defining how the execution context is passed from join points to related advices.

**Precedence constraints.**
- Precedence relationship to specify the execution order between aspects.
**Aspect-UML profile.**

**Semantic annotations.**

---

**Specification of operations.**

- Pre and Post conditions specifying the individual contracts to be met by advices and join points.

---

**Context passing constraints.**

- Relationship defining how the execution context is passed from join points to related advices.

---

**Precedence constraints.**

- Precedence relationship to specify the execution order between aspects.
Aspect-UML profile.

Semantic constraints.
Aspect-UML profile.
Semantic constraints.
Aspect-UML profile.

Semantic constraints.

**context** Connection::drop()
**pre**: status = "connected"
**post**: status = "disconnected"

**context** Connection::merge()
**pre**: status = "disconnected"
**post**: status = "connected"

**aspect** Billing
**getBill**(cust: Customer): Bill
**after** OpDrop(c: Connection) {}

**aspect** Timing
**startTime**: int
**connectionTime**: int
**c**: Connection

**pointCut**
**opComplete**
**call** .*connection.complete()
**OpComplete**(c: Connection)

**pointCut**
**opDrop**
**call** .*connection.drop()
**OpDrop**(c: Connection)
Aspect-UML profile.
Semantic constraints.
Aspect-UML profile.
Semantic constraints.

```
context Connection::drop()
  pre: status = "connected"
  post: status = "disconnected"

context Billing::OpDrop()
  pre: Timing.Timer[c].connectionTime = Null
  post: Billing[c].charge >= Billing[c].charge@pre
```

```
context Timer::OpDrop()
  pre: Timing.Timer[c].connectionTime = Null
  post: Timing.Timer[c].connectionTime != Null
```

```
<aspect>
Timing

getTimer(c:Connection): Timer
  after OpComplete(c:Connection) {}
  after OpDrop(c:Connection) {}
</aspect>
```

```
<aspect>
Billing

getBill(cust: Customer): Bill
  after OpDrop(c:Connection) {}
</aspect>
```
Aspect-UML profile.

Semantic constraints.
Overview of Alloy.

Alloy is a specification language and an analysis tool.

- Uses a notation inspired from Z.
- Uses SAT algorithms for the analysis.
- Limits the analysis to finite (user-bounded) model instances.
Elements of an Alloy specification.

1. Signatures and fields
   - Introduce basic types and a collection of relations.
   - Signature extension supports hierarchical specification.

2. Constraints paragraphs
   - Facts: constraints assumed to hold.
   - Predicates: reusable constraints.
   - Assertions: conjectures to check.

3. Commands
   - run: generates instances of a predicate.
   - check: generates counterexamples to an assertion.
Elements of an Alloy specification.

1. Signatures and fields
   - Introduce basic types and a collection of relations.
   - Signature extension supports hierarchical specification.

2. Constraints paragraphs
   - Facts: constraints assumed to hold.
   - Predicates: reusable constraints.
   - Assertions: conjectures to check.

3. Commands
   - run: generates instances of a predicate.
   - check: generates counterexamples to an assertion.
Elements of an Alloy specification.

1. Signatures and fields
   - Introduce basic types and a collection of relations.
   - Signature extension supports hierarchical specification.

2. Constraints paragraphs
   - Facts: constraints assumed to hold.
   - Predicates: reusable constraints.
   - Assertions: conjectures to check.

3. Commands
   - run: generates instances of a predicate.
   - check: generates counterexamples to an assertion.
Elements of an Alloy specification.

1. Signatures and fields
   - Introduce basic types and a collection of relations.
   - Signature extension supports hierarchical specification.

2. Constraints paragraphs
   - Facts: constraints assumed to hold.
   - Predicates: reusable constraints.
   - Assertions: conjectures to check.

3. Commands
   - run: generates instances of a predicate.
   - check: generates counterexamples to an assertion.

<table>
<thead>
<tr>
<th>Aspect-UML</th>
<th>Alloy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>Signature declaration</td>
</tr>
<tr>
<td>Aspect</td>
<td>Signature declaration</td>
</tr>
<tr>
<td>Attribute, Association</td>
<td>Relation declared within a signature</td>
</tr>
<tr>
<td>Method</td>
<td>Predicate</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Aspect-UML</th>
<th>Alloy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>Signature declaration</td>
</tr>
<tr>
<td>Aspect</td>
<td>Signature declaration</td>
</tr>
<tr>
<td>Attribute, Association</td>
<td>Relation declared within a signature</td>
</tr>
<tr>
<td>Method</td>
<td>Predicate</td>
</tr>
</tbody>
</table>
Translation of Aspect-UML semantic annotations into Alloy.

We use predicates to:

- simulate the execution of operations (advices and join points).
- compose *concurrent* advices.
- weave the composed advices at a given join point.
In the Telecom application.

Predicates are defined for specifying:

- the constraints applying to the drop() method.
- the constraints applying to advices opDropTiming and opDropBilling.
- the constraints imposed by the composition of those advices.
- the constraints induced by the weaving of the composed advice at the drop() join point.
In the Telecom application.

Predicates are defined for specifying:

- the constraints applying to the `drop()` method.
- the constraints applying to advices `opDropTiming` and `opDropBilling`.
- the constraints imposed by the composition of those advices.
- the constraints induced by the weaving of the composed advice at the `drop()` join point.
In the Telecom application.

Predicates are defined for specifying:

- the constraints applying to the `drop()` method.
- the constraints applying to advices `opDropTiming` and `opDropBilling`.
- the constraints imposed by the composition of those advices.
- the constraints induced by the weaving of the composed advice at the `drop()` join point.
In the Telecom application.

Predicates are defined for specifying:

- the constraints applying to the `drop()` method.
- the constraints applying to advices `opDropTiming` and `opDropBilling`.
- the constraints imposed by the composition of those advices.
- the constraints induced by the weaving of the composed advice at the `drop()` join point.
In the Telecom application.

Predicates are defined for specifying:

- the constraints applying to the `drop()` method.
- the constraints applying to advices `opDropTiming` and `opDropBilling`.
- the constraints imposed by the composition of those advices.
- the constraints induced by the weaving of the composed advice at the `drop()` join point.
Constraints applying to operations.
In the Telecom application.

**drop() method**

`pred drop(c, c': Connection) {`  
  `c.status= connected &&`  
  `c'.status=disconnected }`
Constraints applying to operations.
In the Telecom application.

Advice opDrop of aspect Timing

**pred** OpDropTiming(t,t':Timing, c:Connection) {
  t.useTimer[c].connectionTime = Null &&
  t'.useTimer[c].connectionTime ≥ 0
}

Advice opDrop of aspect Billing

**pred** OpDropBilling(b,b':Billing,t:Timing, c:Connection) {
  (t.useTimer[c].connectionTime ≥ 0) &&
  (b.useBill[c.origin.owner].charge ≤
    b'.useBill[c.origin.owner].charge)
}
Constraints for the composition of advices.
In the Telecom application.

Composition of opDropTiming and OpDropBilling advices

\[
\text{pred composition}(b,b':\text{Billing}, t,t':\text{Timing}, c:\text{Connection}) \{ \\
\text{OpDropTiming}(t,t',c) \land \\
\text{OpDropBilling}(b,b',t',c) \lor \\
\text{OpDropBilling}(b,b',t,c) \land \\
\text{OpDropTiming}(t,t',c) \}
\]
Constraints for the weaving of advices.
In the Telecom application.

Weaving of the composed advice at `jp drop()`

```plaintext
pred weaving(b,b':Billing,t,t':Timing, c,c':Connection) {
    drop(c,c') && composition
    (b,b',t,t',c') &&
    (c'.status=disconnected)
}
```
Our verification approach aims to reveal important interference problems:

- Violation of local properties: pre and post conditions.
- Violation of global properties: system invariants.
Verification of local properties:

- Assertions are used to ensure that all initially correct model must satisfy the weaving predicate.

The assertion is of the form:

```plaintext
assert testWeaving{
    all c,c’: Connection, t,t’: Timing, b,b’: Billing |
    initiallyCorrect(c,c’,t,t’,b,b’) => weaving(b,b’,t,t’,c,c’)
}
```

- If the assertion is not valid (within the defined scope) then a counterexample will be generated by the analyzer.
Alloy analysis of Aspect-UML models.

Verification of local properties:

- Assertions are used to ensure that all initially correct model must satisfy the weaving predicate.
- The assertion is of the form:

```plaintext
assert testWeaving{
    all c, c': Connection, t, t': Timing, b, b': Billing |
    initiallyCorrect(c, c', t, t', b, b') => weaving(b, b', t, t', c, c')
}
```

- If the assertion is not valid (within the defined scope) then a counterexample will be generated by the analyzer.
Alloy analysis of Aspect-UML models.

Verification of local properties:

- Assertions are used to ensure that all initially correct model must satisfy the weaving predicate.
- The assertion is of the form:

```
assert testWeaving{
    all c,c': Connection, t,t':Timing, b,b': Billing |
    initiallyCorrect(c,c',t,t',b,b') => weaving(b,b',t,t',c,c') }
```

- If the assertion is not valid (within the defined scope) then a counterexample will be generated by the analyzer.
Use predicates to define invariants

- **An example:** The status of devices engaged in connected connections is busy.

```alloy
pred busyAdvices{
    all d:Device, some c:Connection |
    (d in c.(origin + destination)) && (c.status=connected) =>
    (d.d-status=busy)
}
```
Alloy analysis of Aspect-UML models.
Specification of global properties.

Use predicates to define invariants

- **An example**: The status of devices engaged in connected connections is busy.

**busyAdvices Invariant.**

```
pred busyAdvices{
    all d:Device, some c:Connection |
    (d in c.(origin + destination)) && (c.status=connected) =>
    (d.d-status=busy) }
```
Alloy analysis of Aspect-UML models.
Verification of global properties.

Verification of invariants:

- Invariants are assumed to already hold on the base system.
- Check that they locally hold on the execution trace produced by woven aspects.
  - Alloy provides a facility to handle execution traces, through ordered lists.
Summary.

Our contribution to AO modeling...

- **Aspect-UML profile.**
  - Semantic annotations.

- **Verification using Alloy.**
  - Semantic model verification.
  - Not limited to syntactic detection.
  - Alloy is simple and easy to use.
  - Alloy analysis limits the state explosion problem (bounded scope).
Summary.

Future work:

- Transformation rules.
- Automatic translator from Aspect-UML to Alloy.
- Safety and Liveness properties.
- Compositional Verification.