Design of Mutant Operators for the AspectJ Language

Technical Report
Version 1.0
December/2011

Fabiano Cutigi Ferrari
Computing Department
Federal University of São Carlos (UFSCar) – São Carlos – Brazil
fabiano@dc.ufscar.br

Awais Rashid
Computing Department
Lancaster University – Lancaster – United Kingdom
marash@comp.lancs.ac.uk

José Carlos Maldonado
Computer Systems Department
University of São Paulo (ICMC/USP) – São Carlos – Brazil
jcmaldon@icmc.usp.br
Table of Contents

1 Introduction

2 Group I - Operators for PCDs
2.1 PWSR – Pointcut Weakening by Supertype Replacement .................. 3
2.2 PWIW – Pointcut Weakening by Insertions of Wildcards .................. 4
2.3 PWAR – Pointcut Weakening by Annotations Removal ..................... 6
2.4 PSSR – Pointcut Strengthening by Subtype Replacement ................ 7
2.5 PSWR – Pointcut Strengthening by Wildcard Removal .................... 8
2.6 PSDR – Pointcut Strengthening by Declare Annotations Removal ...... 9
2.7 POPL – Pointcut Weakening or Strengthening by Parameter List Changing 10
2.8 POAC – Pointcut Weakening or Strengthening by After Clause Changing . 11
2.9 POEC – Pointcut Weakening or Strengthening by Exception Throwing Clause Changing .......................................................... 13
2.10 PCTT – Pointcut Changing by this-target Replacement ............... 14
2.11 PCCE – Context Changing by Primitive Pointcut Replacement ........ 15
2.12 PCGS – Pointcut Changing by get-set Replacement ..................... 16
2.13 PCCR – Composition Participant Pointcut Replacement ............... 17
2.14 PCLO – Composition Logical Operators Replacement .................. 18
2.15 PCCC – Control Flow Scope Changing ...................................... 19

3 Group II - Operators for ITDs or other Declarations 20
3.1 DAPC – Aspect Precedence Declaration Changing ...................... 21
3.2 DAPO – Aspect Precedence Declaration Omitting ......................... 22
3.3 DSSR – declare soft Statement Removal .................................. 23
3.4 DEWC – error and warning Declaration Changing .................... 24
3.5 DAIC – Aspect Instantiation Clause Changing ............................. 25

4 Group III - Operators for Advices 26
4.1 ABAR – before-after Advice Kind Replacement .......................... 27
4.2 APSR – proceed Statement Removal ...................................... 28
4.3 APER – proceed Execution Enforcement by Guard Condition Removal 32
4.4 AJSC – Join Point Static Information Source Changing

4.5 ABHA – Behaviour Hindering by Advice Removal

4.6 ABPR – Advice Bound Pointcut Replacement

References
This technical report describes the design of mutation operators for the AspectJ language, originally proposed in our previous research [1]. It includes three groups of operators, each one related to the main programming elements available in AspectJ: pointcut descriptor (PCD) (Chapter 2); intertype declarations (ITD) and other declare-like expressions (Chapter 3); and advices (Chapter 4). The report is currently under elaboration; despite this, it already includes the main characteristics of each proposed operator. Updates will be made available in a regular basis.
Group I - Operators for PCDs

Operators from this group perform syntactic changes in order to:

- Increase the number of join points selected by a PCD;
- Reduce the number of join points selected by a PCD;
- Change the set of join points selected by a PCD;
- Change primitive PCD;
- Change PCD composition rules; and
- Modify PCD clauses which depends on dynamic information.
PCD Weakening Operators

2.1 PWSR – Pointcut Weakening by Supertype Replacement

Description: In a PCD, this operator replaces each Java type or user-defined type present in method patterns, constructor patterns and field patterns with its immediate supertype, if it exists, plus the “+” wildcard.

Constraints: Ascends only one level in the class hierarchy in order to limit the number of generated mutants.

Results: Possible increase in the set of selected join points.

Side-effects: Does not have.

Observations: An immediate supertype may be both a superclass or an implemented interface.

Example: Figure 2.1.

```java
public aspect PWSRGeneric {
  pointcut aPointcut(): call(* ClassA.aMethod(TypeB));
  ... 
}

(b) Original code

public aspect PWSRGenericMutant {
  pointcut aPointcut(): call(* SuperclassOfA+.aMethod(TypeB));
  > pointcut aPointcut(): call(* ClassA.aMethod(SuperclassOfB+));
  ... 
}

(b) PWSR mutants.

Figure 2.1: Example of PWSR application.
2.2 PWIW – Pointcut Weakening by Insertions of Wildcards

Description: In a PCD, this operator inserts wildcards into method patterns, constructor patterns and field patterns. It also replaces modifiers with wildcards.

Details:
1. Replaces each identifier (package, class, method or field) with the “*” wildcard;
2. Replaces the tuple (modifiers + return type) with the “*” wildcard;
3. Inserts the “*” wildcard after each identifier;
4. Inserts the “*” wildcard before each identifier; and
5. Inserts the “+” wildcard at the end of Java types or user-defined types.

Constraints: Does not have.

Results: Possible increase in the set of selected join points.

Side-effects: Currently under analysis.

Example: Figure 2.2.
**public** aspect PWIWGeneric {
    pointcut aPointcut (): call(public aType AClass.aMethod(TypeA, TypeB));
}

(a) Original code

**public** aspect PWIWGenericMutant {
> pointcut aPointcut (): call(public * AClass.aMethod(TypeA, TypeB));
> pointcut aPointcut (): call(public aType *.aMethod(TypeA, TypeB));
> pointcut aPointcut (): call(public aType AClass.*(TypeA, TypeB));
> pointcut aPointcut (): call(* AClass.aMethod(TypeA, TypeB));
> pointcut aPointcut (): call(public aType AClass.aMethod*(TypeA, TypeB));
> pointcut aPointcut (): call(public aType AClass.aMethod(*(TypeA, TypeB)));
> pointcut aPointcut (): call(public aType AClass.aMethod*(TypeA, TypeB+));
> pointcut aPointcut (): call(public aType AClass.aMethod(*(TypeA, TypeB++));
> pointcut aPointcut (): call(public aType+ AClass.aMethod*(TypeA, TypeB));
> pointcut aPointcut (): call(public aType AClass+aMethod*(TypeA, TypeB));
> pointcut aPointcut (): call(public aType AClass.aMethod(*(TypeA++, TypeB+));
> pointcut aPointcut (): call(public aType AClass.aMethod(*(TypeA, TypeB++));
>
}  

(b) PWIW mutants.

Figure 2.2: Example of PWIW application.
2.3 PWAR – Pointcut Weakening by Annotations Removal

**Description:** This operator removes annotation tags from patterns which are present in a PCD.

**Constraints:** If the pattern has a composition of annotations (e.g. `@(A1||A2)`), the full composition is removed.

**Results:** Possible increase in the set of selected join points.

**Side-effects:** Does not have.

**Example:** Figure 2.3.

```java
public aspect PWARGeneric {
    pointcut aPointcut ( ) : call (@(OneAnnotation || AnotherAnnotation) * AClass.*(...));
    ...
}

public aspect PWARGenericMutant {
    > pointcut aPointcut ( ) : call (* AClass.*(...));
    ...
}
```

(a) Original code

(b) PWAR mutants.

**Figure 2.3:** Example of PWAR application.
PCD Strengthening Operators

2.4  PSSR – Pointcut Strengthening by Subtype Replacement

Description: In a PCD, this operator replaces each Java type or user-defined type present in method patterns, constructor patterns and field patterns with their immediate subtypes, if they exist.

Constraints: Descends only one level in the class hierarchy in order to limit the number of generated mutants. Only application-specific (i.e. user-defined) types are considered.

Results: Possible decrease in the set of selected join points.

Side-effects: Does not have.

Observations: If the “+” wildcard is present in the original pattern, it must also be present in respective mutants.

Example: Figure 2.4.

```
public aspect PSSRGeneric {
    pointcut aPointcut (): call(* AClass.aMethod(TypeB));
    ...
}
```

(a) Original code

```
public aspect PSSRGenericMutant {
    pointcut aPointcut (): call(* SubclassOfA.aMethod(TypeB));
    pointcut aPointcut (): call(* AClass.aMethod(SubclassOfB));
    ...
}
```

(b) PSSR mutants.

Figure 2.4: Example of PSSR application.
2.5 PSWR – Pointcut Strengthening by Wildcard Removal

**Description:** In a PCD, this operator removes wildcards from method patterns, constructor patterns and field patterns.

**Details:**

1. Removes the “+” wildcard from the end of Java types and user-defined types; and
2. Removes the “*” wildcard from each identifier (package, class, method or field names).

**Constraints:** The “*” wildcard is not replaced with each class method that belong to the application under test in order to limit the number of generated mutants.

**Results:** Possible decrease in the set of selected join points.

**Side-effects:** Currently under analysis.

**Example:** Figure 2.5.

```
public aspect PSWRGeneric {
    pointcut aPointcut(): call(public aType aPackage*..AClass+.aMethod*(..));
}

(a) Original code

public aspect PSWRGenericMutant {
    > pointcut aPointcut(): call(public aType aPackage*.AClass+.aMethod(..));
    > pointcut aPointcut(): call(public aType aPackage*.AClass+.aMethod*(..));
    > pointcut aPointcut(): call(public aType aPackage*.AClass++.aMethod*(..));
    > pointcut aPointcut(): call(public aType aPackage+.AClass*.aMethod*(..));
}

(b) PSWR mutants.
```

**Figure 2.5:** Example of PSWR application.
2.6 PSDR – Pointcut Strengthening by Declare Annotations Removal

**Description:** This operator removes declare annotations clauses from aspects.

**Constraints:** Does not have.

**Results:** Possible decrease in the set of selected join points.

**Side-effects:** Does not have.

**Example:** Figure 2.6.

```java
public aspect PSDRGeneric {
    declare @method : * AClass.aMethod(...) : @AnAnnotation;
}
```

(a) Original code

```java
public aspect PSDRGeneric {
    // declare @method : * AClass.aMethod(...) : @AnAnnotation;
}
```

(b) PSDR mutants.

**Figure 2.6:** Example of PSDR application.
Chapter 2. Group I - Operators for PCDs

PCD Weakening or Strengthening Operators

2.7 POPL – Pointcut Weakening or Strengthening by Parameter List Changing

Description: In a PCD, this operator replaces, inserts and removes wildcards into/from the parameter list. A parameter list may contain zero or more types, which may be primitive types (e.g. int or float), standard Java types (e.g. String, List or Vector) or user-defined types.

Details:
1. Replaces an empty list with the “..” wildcard;
2. Removes the “..” wildcard from the list;
3. Replaces parameter types with the “..” wildcard;
4. Replaces the “*” wildcard with the “..” wildcard;

Constraints: Does not consider all possibilities for the “..” wildcard. For example, the parameter list (int,int,int) is not mutated to (..,int,int), (int,..,int), (int,int,..), (..,int), (int,..) and (..). This would represent an exhaustive process and should lead to a great number of similar, probably equivalent mutants.

Results: Possible change in the set of selected join points.

Side-effects: Currently under analysis.

Example: Figure 2.7.

```
public aspect POPLGeneric {
    pointcut aPointcut (): call(* AClass.aMethod(TypeA,..,TypeB));
    ...
}                      
(a) Original code

public aspect POPLGeneric {
>    pointcut aPointcut (): call(* AClass.aMethod(TypeA,..,TypeB));
>    pointcut aPointcut (): call(* AClass.aMethod(.,.,TypeB));
>    pointcut aPointcut (): call(* AClass.aMethod(.,..,TypeB));
>    pointcut aPointcut (): call(* AClass.aMethod(.,.,..));
    ...
}                      
(b) POPL mutants.
```

**Figure 2.7**: Example of POPL application.
2.8 POAC – Pointcut Weakening or Strengthening by After Clause Changing

Description: This operator adds, changes and removes the returning or throwing clause from the definition of an advice that runs after a reached joint point.

Details:
1. Adds returning and throwing clauses to an after advice;
2. Removes returning and throwing clauses from an after advice;
3. Replaces a returning clause with a throwing one and vice versa; and
4. Removes declared parameters from returning and throwing clauses.

Constraints: Does not have.

Results: Possible change in the set of selected join points.

Side-effects: This operator may generate non-compilable mutants when the clause to be removed or changed captures context data, since this data might be used or modified within the advice body. Furthermore, this operator may generate syntactically correct mutants that are not able to be woven into the a particular base code due to the base code’s characteristics.

Observations: Although this operator is syntactically related to the definition of advices, it is semantically connected to the number of selected join points. Thus, this operator is included in Group I.

Example: Figure 2.8.
Figure 2.8: Example of POAC application.
2.9 POEC – Pointcut Weakening or Strengthening by Exception Throwing Clause Changing

**Description:** In a PCD, this operator modifies the list of required thrown exceptions.

**Details:**

1. Denies each required exception in the list;
2. Modifies the composition rules of a list of exceptions by:
   - (a) Swaping “and” ("&&") and “or” ("||") logical operators;
   - (b) Replacing the AspectJ-specific “and” composition (represented by a comma) with the “&&” and “||” conventional logical operators; and
3. Removes the full list of required exceptions.

**Constraints:** Does not have.

**Results:** Possible change in the set of selected join points.

**Side-effects:** Does not have.

**Example:** Figure 2.9.

---

### Example: Figure 2.9

```java
public aspect POECGeneric {  
    pointcut aPoincut (): call(* AClass..() throws Except01 || Except02, Except03);  
    ...  
}

(a) Original code

```java
public aspect PCECGenericMutant {  
    > pointcut aPoincut (): call(* AClass..() throws (!Except01) || Except02, Except03);  
    > pointcut aPoincut (): call(* AClass..() throws Except01 || (!Except02), Except03);  
    > pointcut aPoincut (): call(* AClass..() throws Except01 || Except02, (!Except03));  
    > pointcut aPoincut (): call(* AClass..() throws Except01 && Except02, Except03);  
    > pointcut aPoincut (): call(* AClass..() throws Except01 || Except02 && Except03);  
    > pointcut aPoincut (): call(* AClass..() throws Except01 || Except02 || Except03);  
    > pointcut aPoincut (): call(* AClass..());  
    ...  
}

(b) POEC mutants.

Figure 2.9: Example of POEC application.
PCD Changing Operators

2.10  PCTT – Pointcut Changing by this-target Replacement

Description: This operator replaces a this primitive PCD with a target one and vice versa.

Constraints: Does not have.

Results: Possible change in the set of selected join points.

Side-effects: This operator may generate mutants which result in runtime error during any test case execution due to incorrect references to this and target objects obtained from thisJoinPoint object information access. Besides, infinite loops may occur when the join point matching is based on object types and this and target objects are the same.

Example: Figure 2.10.

```java
public aspect PCTTGeneric {
    pointcut aPointcut(AType t): call(* AClass.aMethod(..)) && this(t);
    ...
}

(a) Original code

public aspect PCTTGenericMutant {
>    pointcut aPointcut(AType t): call(* AClass.aMethod(..)) && target(t);
>    ...
}

(b) PCTT mutant.

Figure 2.10: Example of PCTT application.
2.11 PCCE – Context Changing by Primitive Pointcut Replacement

**Description:** This operator swaps call, execution, initialization or preinitialization primitive PCDs.

**Constraints:** initialization and preinitialization are only applicable to constructor-related PCDs.

**Results:** Change in the context under which pieces of advice will run.

**Side-effects:** Currently under analysis.

**Example:** Figure 2.11.

```java
public aspect PCCEGeneric {
    pointcut aPointcut (): (call(AClass.new(...)));
    ...
}
```

(a) Original code

```java
public aspect PCCEGenericMutant {
    pointcut aPointcut (): (execution(AClass.new(...)));
    pointcut aPointcut (): (initialization(AClass.new(...)));
    pointcut aPointcut (): (preinitialization(AClass.new(...)));
    ...
}
```

(b) PCCE mutants.

**Figure 2.11:** Example of PCCE application.
2.12 PCGS – Pointcut Changing by get-set Replacement

**Description:** This operator replaces a `get` primitive pointcut with a `set` one and vice versa.

**Constraints:** Does not have

**Results:** Possible change in the set of selected join points.

**Side-effects:** Does not have.

**Example:** Figure 2.12.

```java
public aspect PCGSGeneric {
    pointcut aPointcut (): get( AType.);
}
...  
```

(a) Original code

```java
public aspect PCGSGenericMutant {
    pointcut aPointcut (): set( AType.);
}
...  
```

(b) PCGS mutant.

**Figure 2.12:** Example of PCGS application.
2.13 PCCR – Composition Participant Pointcut Replacement

Description: This operator replaces pointcuts that are part of a PCD or are used as parameter for matching based on control flow.

Details:
1. Replaces a participant (anonymous or named) of a composed PCD by other named pointcuts defined in the same aspect;
2. Replaces pointcuts used as a parameter in \texttt{cflow} or \texttt{cflowbelow} expressions with other named pointcuts defined in the same aspect.

Constraints: Pointcuts that are source for replacements should not include context exposure, otherwise this would probably result in non-compilable mutants. In addition, this operator is only applied to pointcuts that do not involve context exposure. When applied to \texttt{cflow} and \texttt{cflowbelow} expressions, this operator is not applied recursively. That is, in case a \texttt{cflow} or a \texttt{cflowbelow} surrounds another \texttt{cflow} or \texttt{cflowbelow}, the internal ones are not considered.

Results: Possible change in the set of selected join points.

Side-effects: Currently under analysis.

Example: Figure 2.13.

\begin{verbatim}

\textbf{public} aspect PCCRGeneric { 
  pointcut pointcutA(): ... ; 
  pointcut pointcutB(): ... ; 
  pointcut pointcutC(): ... ; 
  pointcut compPointcut (): pointcutA() && pointcutB(); 
  ... 
}

(a) Original code

\textbf{public} aspect PCCRGenericMutant { 
  ... 
  > pointcut compPointcut (): pointcutA() && pointcutA(); 
  > pointcut compPointcut (): pointcutA() && pointcutC(); 
  > pointcut compPointcut (): pointcutB() && pointcutB(); 
  > pointcut compPointcut (): pointcutC() && pointcutB(); 
  ... 
}

(b) PCCR mutants.

\end{verbatim}

\textbf{Figure 2.13:} Example of PCCR application.
2.14 PCLO – Composition Logical Operators Replacement

Description: This operator changes logical operators present in type composition rules and PCD composition rules.

Details:
1. Replaces “&&” operators with “||” ones and vice versa;
2. Adds the “!” operator to each composition participant; and
3. Removes the “!” operator from compositions.

Constraints: Does not consider compositions involving the following primitive PCD: target, this and args.

Results: Possible change in the set of selected join points.

Side-effects: Currently under analysis.

Example: Figure 2.14.

```
public aspect PCLOGeneric {
    pointcut pointcutA(): ...;
    pointcut pointcutB(): ...;
    pointcut compPointcut(): pointcutA() || pointcutB();
} ...  

(a) Original code

public aspect PCLOGenericMutant {
    pointcut compPointcut(): pointcutA() && pointcutB();
    pointcut compPointcut(): !pointcutA() || pointcutB();
    pointcut compPointcut(): pointcutA() || !pointcutB();
} ...  

(b) PCLO mutants.
```

Figure 2.14: Example of PCLO application.
2.15 PCCC – Control Flow Scope Changing

**Description:** This operator replaces a `cflow` primitive pointcut with a `cflowbelow` one and vice versa.

**Constraints:** Does not have.

**Results:** Possible change in the set of advice executions.

**Side-effects:** Does not have.

**Example:** Figure 2.15.

```
public aspect PCCCGeneric {
    pointcut pointcutA(): ... ;
    pointcut pointcutB(): ... ;

    pointcut comp(): pointcutA() && cflow(pointcutB());
    ...
}

(a) Original code

public aspect PCCCGenericMutant {
    ...
    > pointcut comp(): pointcutA() && cflowbelow(pointcutB());
    ...
}

(b) PCCC mutant.
```

**Figure 2.15:** Example of PCCC application.
Group II - Operators for ITDs or other Declarations

Operators from this group perform syntactic changes in order to:

- Modify the precedence among aspects;
- Alter exceptions severity;
- Remove error and warning declarations;
- Modify rules related to aspect instantiation;
3.1 DAPC – Aspect Precedence Declaration Changing

**Description:** This operator alternates the precedence among two or more aspects.

**Constraints:** Does not have.

**Results:** Possible unintended execution order of advices which advise common join points.

**Side-effects:** Currently under analysis.

**Observations:** It may consider all precedence possibilities among involved aspects. However, the number $n$ of mutants may be too large, in the order of $n!$. An option is to constrain the maximum number of elements (e.g. 4 or 5 aspects) in the `declare precedence` expression.

**Example:** Figure 3.1.

```java
public aspect DAPCGeneric {
    declare precedence: AspectA, AspectB, AspectC;
    ...
}
(a) Original code

public aspect DAPCGenericMutant {
>    declare precedence: AspectA, AspectC, AspectB;
>    declare precedence: AspectB, AspectA, AspectC;
>    declare precedence: AspectB, AspectC, AspectA;
>    declare precedence: AspectC, AspectA, AspectB;
>    declare precedence: AspectC, AspectB, AspectA;
    ...
}
(b) DAPC mutants.
```

**Figure 3.1:** Example of DAPC application.
3.2 DAPO – Aspect Precedence Declaration Omitting

**Description:** This operator removes aspect precedence declarations.

**Constraints:** Does not have.

**Results:** Arbitrary execution order of advices which advise common join points.

**Side-effects:** Does not have.

**Example:** Figure 3.2.

```java
public aspect DAPOGeneric {  
    declare precedence: AspectA, AspectB;
  } ...

(a) Original code

public aspect DAPOGenericMutant {  
  ->  // declare precedence: AspectA, AspectB;
  } ...

(b) DAPO mutants.
```

**Figure 3.2:** Example of DAPO application.
3.3 DSSR – declare soft Statement Removal

**Description:** This operator removes `declare soft` statements.

**Constraints:** Does not have.

**Results:** Possible unintended control flow execution since previously softened exceptions did not require exception handling implementation.

**Side-effects:** This operator may generate non-compilable mutants when the signaled exception is not properly handled through exception flow implementation.

**Example:** Figure 3.3.

```java
public aspect DSSRGeneric {
    ...
    declare soft: anException : aPointcut;
    ...
}
```

(a) Original code

```java
public aspect DSSRGenericMutant {
    ...
    -> // declare soft: anException : aPointcut;
    ...
}
```

(b) DSSR mutant.

**Figure 3.3:** Example of DSSR application.
3.4 DEWC – error and warning Declaration Changing

Description: This operator modifies error and warning declarations.

Details:
1. Removes error and warning declarations, one at a time; and
2. Replaces error declarations with warnings and vice versa.

Constraints: Does not have

Results: Possible execution of unintended control flow.

Side-effects: This operator may generate non-compilable mutants when a warning declaration is replaced with an error one and the PCD bound to these declarations matches at least one join point in the base code. When the PCD matches no join points (e.g. the policy enforcement rules are strictly followed), this kind of replacement results in compilable mutants.

Example: Figure 3.4.

```
public aspect DEWCGeneric {
    pointcut pointcutA(); ... ;
    declare error: pointcutA(): "a message...";
} ...

(a) Original code
```

```
public aspect DEWCGeneric {
    pointcut pointcutA(); ... ;
    // declare error: pointcutA(): "a message...";
    declare warning: pointcutA(): "a message...";
} ...

(b) DEWC mutants.
```

Figure 3.4: Example of DEWC application.
3.5 DAIC – Aspect Instantiation Clause Changing

Description: This operator modifies aspect instantiation clauses.

Details:
1. Replaces an aspect instantiation clause by another, varying among `perthis`, `pertarget`, `perflow` and `perflowbelow`; and
2. Removes the existing clause, making the aspect singleton.

Constraints: Does not have.

Results: Possible incorrect objects and/or aspect states. For example, incorrect values gathered from an aspect may be used in the definition of an object state, resulting in similar problem for both entities.

Side-effects: Does not have.

Example: Figure 3.5.

```
public aspect DAICGeneric pertarget(aPointcut) {
    ...
}

(a) Original code

> public aspect DAICGenericMutant perthis(aPointcut) {
>     ...
> }
> public aspect DAICGenericMutant perflow(aPointcut) {
>     ...
> }
> public aspect DAICGenericMutant perflowbelow(aPointcut) {
>     ...
> }
> public aspect DAICGenericMutant {
>     ...
> }

(b) DAIC mutants.
```

Figure 3.5: Example of DAIC application.
Group III - Operators for Advices

Operators from this group perform syntactic changes in order to:

- Alter the moment at which an advice is executed;
- Deactivate and activate the execution of the reached join point;
- Alter the origin of static information used within an advice;
- Deny the execution of an advice by its removal; and
- Replace the PCD bound to an advice.
4.1 ABAR – before-after Advice Kind Replacement

**Description:** This operator alters the moment when an advice runs, varying between before and after possibilities.

**Details:**
1. Replaces a before advice kind with after, after returning and after throwing;
2. Replaces an after, after returning or after throwing advice kinds with a before.

**Constraints:** Does not have.

**Results:** Possible failures and state inconsistencies due to changes in the execution sequence of join points and advices.

**Side-effects:** This operator may generate non-compilable mutants when the candidate after advice captures context data, since such data may be used or modified within the advice body.

**Example:** Figure 4.1.

```
public aspect ABARGeneric {
    pointcut aPointcut (): (call(* AClass.aMethod(. . .))) ;

    before (): aPointcut () {
    ... 
    }
}
```

(a) Original code

```
public aspect ABARGeneric {
    pointcut aPointcut (): (call(* AClass.aMethod(. . .))) ;

    after (): aPointcut () {
    after () returning: aPointcut ()
    after () throwing: aPointcut () {
    ... 
    }
}
```

(b) ABAR mutants.

**Figure 4.1:** Example of ABAR application.
4.2 APSR – proceed Statement Removal

Description: This operator removes occurrences of the `proceed` statement, one by one. Three possible situations are following discussed.

Details:
1. When an execution of the `proceed` statement is not expected to return any value, such occurrence can be directly removed.
2. When a variable or object reference is defined from the value returned from a `proceed` call, we should consider two possibilities:

   (i) If the returning value is an object reference (also called *complex type*), the occurrence is replaced with a new object instance of the same type. Besides, it is replaced by the required constants presented in Table 4.1.

   (ii) If the returning value is a primitive type (e.g. `int`, `float` or `boolean`), the occurrence is replaced with the required constants presented in Table 4.1, adapted from the work of Vincenzi [2].

<table>
<thead>
<tr>
<th>Variable type</th>
<th>Required constants</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>true, false</td>
</tr>
<tr>
<td>byte</td>
<td>-1, 1, 0, Byte.MAX_VALUE, Byte.MIN_VALUE</td>
</tr>
<tr>
<td>char</td>
<td>1, Character.MAX_VALUE, Character.MIN_VALUE</td>
</tr>
<tr>
<td>double</td>
<td>-1.0d, 1.0d, 0.0d, Double.MAX_VALUE, Double.MIN_VALUE</td>
</tr>
<tr>
<td>float</td>
<td>-1.0f, 1.0f, 0.0f, Float.MAX_VALUE, Float.MIN_VALUE</td>
</tr>
<tr>
<td>int</td>
<td>-1, 1, 0, Integer.MAX_VALUE, Integer.MIN_VALUE</td>
</tr>
<tr>
<td>long</td>
<td>-1L, 1L, 0L, Long.MAX_VALUE, Long.MIN_VALUE</td>
</tr>
<tr>
<td>short</td>
<td>-1, 1, 0, Short.MAX_VALUE, Short.MIN_VALUE</td>
</tr>
<tr>
<td>Complex types</td>
<td>null, new object instance</td>
</tr>
</tbody>
</table>

3. When a `proceed` statement is used in the return statement of an advice, the same kinds of mutation described in Case 2 above are applicable.

Constraints: Does not consider array types generally (e.g. `int[]`, `Vector[]` etc), since array sizes should be defined in advance.

Results: Behaviour execution partially hindered, probably leading to incorrect states and results.

Side-effects: Currently under analysis.

Example: Figure 4.2 (case 1), Figure 4.3 (case 2(i)), Figure 4.4 (case 2(ii)), Figure 4.5 (case 3(i)) and Figure 4.6 (case 3(ii)).
Chapter 4. Group III - Operators for Advices

(a) Original code

public aspect APSR01Generic {
    pointcut aPointcut() : (call(* AClass.aMethod(..)));

    void around(): aPointcut() {
        ...
        proceed();
        ...
    }
}

(b) APSR mutants

Figure 4.2: Example of APSR application - Case 1.

(a) Original code

public aspect APSR02Generic {
    pointcut aPointcut() : (call(* AClass.aMethod(..)));

    AType around(): aPointcut() {
        ...
        AType x = proceed();
        ...
    }
}

(b) APSR mutants

Figure 4.3: Example of APSR application - Case 2(i).
**Figure 4.4:** Example of APSR application - Case 2(ii).

**Figure 4.5:** Example of APSR application - Case 3(i).
public aspect APSR03Generic {
    pointcut aPointcut(): (call(∗ AClass.aMethod(..)))
    aPrimitiveType around(): aPointcut() {
        ...
        return proceed();
    }
    ...
}

(a) Original code

public aspect APSR03GenericMutant {
    pointcut aPointcut(): (call(∗ AClass.aMethod(..)))
    aPrimitiveType around(): aPointcut() {
        ...
        > return aRequiredConstant;
    }
    ...
}

(b) APSR mutants

Figure 4.6: Example of APSR application - Case 3(ii).
4.3 APER – proceed Execution Enforcement by Guard Condition Removal

Description: This operator enables the execution of proceed statements by removing the guard conditions that surround them.

Details:

1. Applies to if-else and switch structures. If a proceed is placed inside a sequence of nested if’s, the operator is applied to all of them, one by one incrementally.
2. Specifically for else and switch structures, the full block of statements\(^1\) in which the proceed statement appears is moved out of the respective structure, except the break command.

Constraints: If a target switch block contains more than one break command, this block is not considered for mutation since an in-depth analysis of the respective piece of code would be required. In addition, the mutation is performed if and only if the break is unlabelled.

Results: Possible execution of unintended behaviour, implemented both in the original code or in advices, which may lead to inconsistent state and incorrect results.

Side-effects: This operator may generate non-compilable mutants if variables are defined in a target if or switch condition clause.

Observations:

1. In order to avoid compilation errors (for example, when there is a return statement inside the same block) the block of instructions is place inside a if(true) structure.
2. In cases the proceed statement appears repeatedly inside the same block, all occurrences are moved together (only one mutant is created).

Example: Figures 4.7 (if mutated), 4.8 (else mutated) and 4.9 (switch mutated).

\(^1\)A block of statements (or commands) is informally defined as a set of statements which are executed sequentially without in and out branches.
public aspect APER01Generic {
    ...
    void around(): aPointcut () {
        if (aPredicate)
            proceed();
        ...
    }
}

(a) Original code

public aspect APER01GenericMutant {
    void around(): aPointcut () {
        if (true)
            proceed();
        ...
    }
}

(b) APER mutants.

Figure 4.7: Example of APER application - if mutated.

public aspect APER01Generic {
    ...
    void around(): aPointcut () {
        if (aPredicate) { ... }
        else {
            ...possibly some commands...
            proceed();
            ...possibly other commands...
        }
    }
}

(a) Original code

public aspect APER01GenericMutant {
    ...
    void around(): aPointcut () {
        if (aPredicate) { ... }
        else {
            // ...possibly some commands...
            // proceed();
            // ...possibly other commands...
        }
        if (true) {
            ...possibly some commands...
            proceed();
            ...possibly other commands...
        }
    }
}

(b) APER mutants.

Figure 4.8: Example of APER application - else mutated.
public aspect APER02Generic {
    ... 
    void around (): aPointcut () { 
        switch (aVariable) {
            case someValue: 
                ... possibly some commands
                proceed (); 
                ... possibly other commands
                break;
            case anotherValue:
                ... 
        }
    }
}

(a) Original code

public aspect APER02GenericMutant {
    ... 
    void around (): aPointcut () { 
        switch (aVariable) {
            case someValue:
                ... possibly some commands
                proceed (); 
                ... possibly other commands
                break;
            case anotherValue:
                ... 
        }
        if (true) {
            // possibly some commands
            proceed ();
            // possibly other commands
        }
    }
}

(b) APER mutants.

Figure 4.9: Example of APER application - switch mutated.
4.4 AJSC – Join Point Static Information Source Changing

Description: This operator replaces a `thisJoinPointStaticPart` with a `thisEnclosingJoinPointStaticPart` object reference and vice versa. Such modification changes the origin of static information gathered from a join point that can be used within an advice.

Constraints: Does not have.

Results: Possible accesses to incorrect static information inside an advice.

Side-effects: Does not have.

Example: Figure 4.10.

```java
public aspect AJSCGeneric {
    ...
    anAdviceType(): aPointcut() {
        ...
        System.out.println("info: " + thisJoinPointStaticPart.amethod());
        ...
    }
}

(a) Original code

public aspect AJSCGenericMutant {
    ...
    anAdviceType(): aPointcut() {
        ...
        System.out.println("info: " + thisEnclosingJoinPointStaticPart.amethod());
        ...
    }
}

(b) AJSC mutants.
```

Figure 4.10: Example of AJSC application.
4.5 ABHA – Behaviour Hindering by Advice Removal

Description: This operator removes implemented advices, one by one, hindering the execution of crosscutting behaviour.

Constraints: Does not have.

Results: Expected behaviour probably not executed.

Side-effects: Does not have.

Example: Figure 4.11.

```java
public aspect ABHAGeneric {
    ...
    before(): aPointcut() { ... }
    ...
}

(a) Original code

public aspect ABHAGenericMutant {
    ...
    // before(): aPointcut() { ... }
    ...
}

(b) ABHA mutants.
```

Figure 4.11: Example of ABHA application.
4.6 ABPR – Advice Bound Pointcut Replacement

**Description:** This operator replaces PCDs that are bound to an advice with other PCDs defined in the same aspect.

**Details:**

1. Replaces the right side of an advice definition (i.e the full PCD bound to this advice) with another PCD bound to other advices of the same aspect;

**Constraints:** Does not have.

**Results:** Possible execution of behaviour in unintended join points.

**Side-effects:** This operator may generate non-compilable mutants in cases which the PCD to be replaced involves context exposure.

**Example:** Figure 4.12.

```
public aspect ABPRGeneric {
    before () : pointcutA() { ... }
    after () : pointcutB() { ... }
}

(b) Original code

public aspect ABPRGenericMutant {
> before () : pointcutB() { ... }
> after () : pointcutA() { ... }
}

(b) ABPR mutants.
```

**Figure 4.12:** Example of ABPR application.
References
